Using
ADOBE® FLEX® 4.6

Legal notices

Contents

Chapter 1: Introduction to Flex 4.6

Chapter 2: Getting started

Chapter 3: Application architecture

Chapter 4: Building the user interface

Chapter 5: Using data-driven UI components

Chapter 6: Enhancing the user interface

Chapter 7: Enhancing usability

Chapter 8: Developer tools

Chapter 9: Testing and automation

Chapter 10: Custom components

Chapter 11: Deploying applications

Chapter 1: Introduction to Flex 4.6

Adobe® Flex® 4.6 introduces new features and continues to build upon the major architectural changes in Flex 4.

If you are a new Flex developer, and have not previously developed applications in Flex, you can use the many available resources to learn more about Flex. For more information on the resources available for new Flex developers, see "Getting started with Flex 4.6" on page 1 and What's new in Flex 4.6 SDK.

If you are an existing Flex developer, you can use the information in "Migration" on page 2 to decide how best to upgrade your applications to the newest SDK.

Getting started with Flex 4.6

Adobe and the Flex developer community provide many resources that you can use to get started with Flex development. These resources are helpful for new users, and for experienced users seeking to increase their knowledge of Flex.

The Flex Developer Center contains many resources that you can help you start using Flex 4.6, including:

- Getting Started articles, links and tutorials
- Samples of real applications built in Flex
- The Flex Cookbook, which contains answers to common coding problems
- Links to the Flex community and to other sites devoted to Flex

As a new Flex user, there are other Adobe sites that you can also use to get familiar with Flex, including the following:

- Adobe Flex in a Week at http://www.adobe.com/devnet/flex/videotraining/
- Flex Test Drive at http://www.adobe.com/devnet/flex/testdrive.html
- Flex video training at Adobe Flex TV

Features new for Flex 4.6

Flex 4.6 contains several new features, including:

- More Spark mobile components including: SplitViewNavigator, CalloutButton, Callout, SpinnerList, DateSpinner, and ToggleSwitch
- Better performance
- Updated platform support
- Enhanced Tooling – Flash Builder 4.6
- Text Enhancements

For more information on additional features, see What's new in Flex 4.6 SDK and What's New in Flash Builder 4.6.

Features new for Flex 4.5

Flex 4.5 contains several new features, including:

- Support for mobile applications
- More Spark UI components including: Image, DataGrid, and Form
- Integration with Flash Player’s globalization classes
- RSL enhancements
- TLF 2.0
- OSMF 1.0 integration
- Spark validators and formatters

This list is a subset of the new features in Flex 4.5 SDK. For more information on additional features, see What's New in Flex 4.5 SDK. You can get more information about productivity enhancements in Flash Builder 4.5 at What's New in Flash Builder 4.5.

Development tools for Flex 4.6

Flex developers typically use two development tools:

- Adobe® Flash® Builder™

  Flash Builderis an integrated development environment (IDE) for building cross-platform, rich Internet applications (RIAs). Using Flash Builder, you build applications that use the Adobe Flex framework, MXML, Adobe Flash Player, Adobe AIR, ActionScript 3.0, Adobe® LiveCycle® Data Services ES, and the Adobe Flex Charting components. Flash Builder also includes testing, debugging, and profiling tools that lead to increased levels of productivity and effectiveness.

  For more information on Flash Builder, see About Flash Builder.

- Adobe® Flash® Catalyst™

  Catalyst makes it easy for designers to create Flex-based RIA UI's from artwork imported from Adobe® Creative Suite® tools, and to define interactions and behaviors within Catalyst. The applications created in Catalyst are Flex applications. Flash Builder provides a simple workflow for importing these applications, which allows designers to collaborate with developers more easily than ever before.

  For more information on Flash Catalyst, see About Flash Catalyst.

Migration

If you are an existing Flex customer, you might migrate your existing applications from Flex 4 to Flex 4.6, or even from Flex 3.

For Flex 4 users, migrating to Flex 4.6 is relatively simple. The differences are largely cosmetic and are summarized in Flex Backwards Compatibility.

For Flex 3 users, before starting the migration process, you should be aware of all new Flex 4 and 4.6 features, and be familiar with changes to existing features. The greatest differences are from Flex 3 to Flex 4. For information on migrating applications from Flex 3 to Flex 4, see the *Adobe Flex 4 Features and Migration Guide*at http://www.adobe.com/go/learn_flex4_featuremigrate_en.

If you upgrade to a new version of Flash Builder to take advantage of new features in the IDE but do not want to use the new compiler features, you can downgrade the output application. You do this by setting the `player-version` compiler option. You can also select an older SDK to compile against. For more information, see “Backward compatibility” on page 2239.

Chapter 2: Getting started

Developing applications in MXML

MXML is an XML language that you use to lay out user interface components for applications built in Adobe® Flex®. You also use MXML to declaratively define nonvisual aspects of an application, such as access to server-side data sources and data bindings between user interface components and server-side data sources.

For information on MXML syntax, see "MXML syntax" on page 21.

About MXML

You use two languages to write applications in Flex: MXML and ActionScript. MXML is an XML markup language that you use to lay out user interface components. You also use MXML to declaratively define nonvisual aspects of an application, such as access to data sources on the server and data bindings between user interface components and data sources on the server.

Like HTML, MXML provides tags that define user interfaces. MXML will seem very familiar if you have worked with HTML. However, MXML is more structured than HTML, and it provides a much richer tag set. For example, MXML includes tags for visual components such as data grids, trees, tab navigators, accordions, and menus, as well as nonvisual components that provide web service connections, data binding, and animation effects. You can also extend MXML with custom components that you reference as MXML tags.

One of the biggest differences between MXML and HTML is that MXML-defined applications are compiled into SWF files and rendered by Adobe® Flash® Player or Adobe® AIR™, which provides a richer and more dynamic user interface than page-based HTML applications.

You can write an MXML application in a single file or in multiple files. MXML also supports custom components written in MXML and ActionScript files.

Using Spark and MX component sets

Flex defines two sets of components: MX and Spark. The MX component set was included in previous releases of Flex, and is defined in the mx.* packages. The Spark component set is new for Flex 4 and is defined in the spark.* packages. The Spark components use a new architecture for skinning and have other advantages over the MX components.

The MX and Spark component sets contain many of the same components. For example, both component sets defines a Button control, TextInput control, and List control. However, while you can use MX components to perform most of the same actions that you can perform by using the Spark components, Adobe recommends that you use the Spark components when possible.

Writing a simple application in MXML

Because MXML files are ordinary XML files, you have a wide choice of development environments. You can write MXML code in a simple text editor, a dedicated XML editor, or an integrated development environment (IDE) that supports text editing. Flex supplies a dedicated IDE, called Adobe® Flash™ Builder™, that you can use to develop your applications.

The following example shows a simple “Hello World” application that contains just an `<s:Application>` tag and three child tags, the `<s:Panel>` tag and the `<s:Label>` tags, plus a `<s:layout>` tag. The `<s:Application>` tag defines the Application container that is always the root tag of an application. The `<s:Panel>` tag defines a Panel container that includes a title bar, a title, a status message, a border, and a content area for its children. The `<s:Label>` tag represents a Label control, a very simple user interface component that displays text.

```
<?xml version="1.0"?>
<!-- mxml\HellowWorld.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="My Application">
        <s:Label text="Hello World" fontWeight="bold" fontSize="24"/>
    </s:Panel>
</s:Application>
```

Save this code to a file named hello.mxml. MXML filenames must end in a lowercase .mxml file extension.

About XML encoding

The first line of the document specifies an optional declaration of the XML version. It is good practice to include encoding information that specifies how the MXML file is encoded. Many editors let you select from a range of file encoding options. On North American operating systems, ISO-8859-1 is the dominant encoding format, and most programs use that format by default. You can use the UTF-8 encoding format to ensure maximum platform compatibility. UTF-8 provides a unique number for every character in a file, and it is platform-, program-, and language-independent.

If you specify an encoding format, it must match the file encoding you use. The following example shows an XML declaration tag that specifies the UTF-8 encoding format:

```
<?xml version="1.0" encoding="utf-8"?>
```

About the <s:Application> tag

In addition to being the root tag of an application, the `<s:Application>` tag represents a Spark Application container. A *container* is a user-interface component that contains other component sets, and uses layout rules for positioning its child components. By default, the Spark Application container lets that you set the position of its children. In the previous example, you set the layout of the container to VerticalLayout so that the Application container automatically lays out its children in a vertical column.

You can nest other types of containers inside an Application container, such as the Panel container shown above, to position user interface components according to other rules. For more information, see “Visual components” on page 280.

About namespaces

In an XML document, tags are associated with a namespace. XML namespaces let you refer to more than one set of XML tags in the same XML document. The `xmlns` property in an MXML tag specifies an XML namespace.

In Flex, you typically define three namespaces:

- `xmlns:fx="http://ns.adobe.com/mxml/2009"` The namespace for top-level ActionScript elements, such as Object and Array, and for tags built into the MXML compiler, such as `<fx:Script>`.
- `xmlns:mx="library://ns.adobe.com/flex/mx"` The namespace for the MX component set.

- `xmlns:s="library://ns.adobe.com/flex/spark"` The namespace for the Spark component set.

In general, you include the Spark and MX component namespaces so that you can use any components from those sets. Where possible, use the Spark components. However, not all MX components have Spark counterparts, so the components in the MX namespace are also sometimes necessary.

You can define additional namespaces for your custom component libraries. For more information on namespaces, see “Using XML namespaces” on page 10.

About MXML tag properties

The properties of an MXML tag, such as the `text`, `fontWeight`, and `fontSize` properties of the `<s:Label>` tag, let you declaratively configure the initial state of the component. You can use ActionScript code in an `<fx:Script>` tag to change the state of a component at run time. For more information, see “Using ActionScript” on page 32.

Compiling MXML to SWF Files

If you are using Flash Builder, you compile and run the compiled SWF file from within Flash Builder. After your application executes correctly, you deploy it by copying it to a directory on your web server or application server.

You can deploy your application as a compiled SWF file, as a SWF file included in an AIR application or, if you have Adobe LiveCycle Data Services ES, you can deploy your application as a set of MXML and AS files.

End users of the application do not typically reference the SWF file directly in an HTTP request. Instead, you embed the application SWF file in an HTML page. The HTML page then uses a script to load the SWF file. Collectively, the HTML page and the script are known as the *wrapper*.

When the SWF file is embedded in the HTML page, users then access the deployed SWF file by making an HTTP request to the HTML page, in the form:

```
http://hostname/path/filename.html
```

For more information on wrappers, see “Creating a wrapper” on page 2552.

Flex also provides a command-line MXML compiler, mxmlc, that lets you compile MXML files. You can use mxmlc to compile hello.mxml from a command line, as the following example shows:

```
cd flex_install_dir/bin
mxmlc --show-actionscript-warnings=true --strict=true c:/app_dir/hello.mxml
```

In this example, *flex_install_dir* is the Flex installation directory, and *app_dir* is the directory containing hello.mxml. The resultant SWF file, hello.swf, is written to the same directory as hello.mxml.

For more information about mxmlc, see “Flex compilers” on page 2164.

The relationship of MXML tags to ActionScript classes

Adobe implemented Flex as an ActionScript class library. That class library contains components (containers and controls), manager classes, data-service classes, and classes for all other features. You develop applications by using the MXML and ActionScript languages with the class library.

MXML tags correspond to ActionScript classes or properties of classes. Flex parses MXML tags and compiles a SWF file that contains the corresponding ActionScript objects. For example, Flex provides the ActionScript Button class that defines the Flex Button control. In MXML, you create a Button control by using the following MXML statement:

```
<s:Button label="Submit"/>
```

When you declare a control using an MXML tag, you create an instance of that class. This MXML statement creates a Button object, and initializes the `label` property of the Button object to the string `"Submit"`.

An MXML tag that corresponds to an ActionScript class uses the same naming conventions as the ActionScript class. Class names begin with an uppercase letter, and uppercase letters separate the words in class names. Every MXML tag attribute corresponds to a property of the ActionScript object, a style applied to the object, or an event listener for the object. For a complete description of the Flex class library and MXML tag syntax, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Understanding the structure of an application built with Flex

You can write an MXML application in a single file or in multiple files. You typically define a main file that contains the `<s:Application>` tag. From within your main file, you can then reference additional files written in MXML, ActionScript, or a combination of the two languages.

A common coding practice is to divide your Flex application into functional units, or modules, where each module performs a discrete task. In Flex, you can divide your application into separate MXML files and ActionScript files, where each file corresponds to a different module. By dividing your application into modules, you provide many benefits, including the following:

Ease of development Different developers or development groups can develop and debug modules independently of each other.

Reusability You can reuse modules in different applications so that you do not have to duplicate your work.

Maintainability You can isolate and debug errors faster than if your application is developed in a single file.

In Flex, a module corresponds to a custom component implemented either in MXML or in ActionScript. These custom components can reference other custom components. There is no restriction on the level of nesting of component references in Flex. You define your components as required by your application.

You can also use sub-applications rather than modules to develop applications that are not monolithic.

Developing applications

MXML development is based on the same iterative process used for other types of web application files such as HTML, JavaServer Pages (JSP), Active Server Pages (ASP), and ColdFusion Markup Language (CFML). Developing a useful Flex application is as easy as opening your favorite text editor, typing some XML tags, saving the file, requesting the file's URL in a web browser, and then repeating the same process.

Flex also provides tools for code debugging. For more information, see "Command-line debugger" on page 2209.

Laying out a user interface using containers

In the Flex model-view design pattern, user interface components represent the view. The MXML language supports two types of user interface components: controls and containers. Controls are form elements, such as buttons, text fields, and list boxes. Containers are rectangular regions of the screen that contain controls and other containers.

You use container components for laying out a user interface, and for controlling user navigation through the application. Examples of layout containers include the HGroup container for laying out child components horizontally and the VGroup container for laying out child components vertically. Examples of navigator containers include the MX TabNavigator container for creating tabbed panels and the MX Accordion navigator container for creating collapsible panels. Typical properties of a container tag include `id`, `width`, and `height`. For more information about the standard Flex containers, see "Introduction to containers" on page 326.

The following example application contains a Spark List control on the left side of the user interface and an MX TabNavigator container on the right side. Both controls are enclosed in a Spark Panel container:

```
<?xml version="1.0"?>
<!-- mxml/LayoutExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="My Application">
        <s:HGroup>
            <!-- List with three items -->
            <s:List>
                <s:dataProvider>
                    <mx:ArrayCollection>
                        <fx:String>Item 1</fx:String>
                        <fx:String>Item 2</fx:String>
                        <fx:String>Item 3</fx:String>
                    </mx:ArrayCollection>
                </s:dataProvider>
            </s:List>
            <!-- First pane of TabNavigator -->
            <mx:TabNavigator borderStyle="solid">
                <s:NavigatorContent label="Pane1" width="300">
                    <s:layout>
                        <s:VerticalLayout/>
                    </s:layout>
                    <s:TextArea text="Hello World"/>
                    <s:Button label="Submit"/>
                </s:NavigatorContent>
                <!-- Second pane of TabNavigator -->
                <s:NavigatorContent label="Pane2" width="300" height="150">
                    <!-- Stock view goes here -->
                </s:NavigatorContent>
            </mx:TabNavigator>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

The List control and TabNavigator container are laid out side by side because they are in an HGroup container. The controls in the TabNavigator container are laid out from top to bottom because they are in a NavigatorContent containers that use the VerticalLayout class.

For more information about laying out user interface components, see “Visual components” on page 280.

Adding user interface controls

Flex includes a large selection of user interface components, such as Button, TextInput, and ComboBox controls. After you define the layout and navigation of your application by using container components, you add the user interface controls.

The following example contains an HGroup (horizontal group) container with two child controls, a TextInput control and a Button control. An HGroup container lays out its children horizontally.

```
<?xml version="1.0"?>
<!-- mxml/AddUIControls.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            private function storeZipInDatabase(s:String):void {
                // event handler code here
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:TextInput id="myText"/>
        <s:Button click="storeZipInDatabase(myText.text);"/>
    </s:HGroup>
</s:Application>
```

Typical properties of a control tag include `id`, `width`, `height`, `fontSize`, `color`, event listeners for events such as `click` and `change`, and effect triggers such as `showEffect` and `rollOverEffect`. For information about the standard Flex controls, see "UI Controls" on page 643.

Using the id property with MXML tags

With a few exceptions (see "MXML tag rules" on page 32), an MXML tag has an optional `id` property, which must be unique within the MXML file. If a tag has an `id` property, you can reference the corresponding object in ActionScript.

The following example uses the `trace()` function to write the value of the `text` property of a TextInput control to the log file:

```
<?xml version="1.0"?>
<!-- mxml/UseIDProperty.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            private function writeToLog():void {
                trace(myText.text);
            }
        ]]>
    </fx:Script>
    <s:VGroup id="myVGroup">
        <s:TextInput id="myText"
            text="Hello World!" />
        <s:Button id="mybutton"
            label="Get Weather"
            click="writeToLog();"/>
    </s:VGroup>
</s:Application>
```

This code causes the MXML compiler to generate a public variable named `myText` that contains a reference to the TextInput instance. This variable lets you access the component instance in ActionScript. You can explicitly refer to the TextInput control's instance with its `id` instance reference in any ActionScript class or script block. By referring to a component's instance, you can modify its properties and call its methods.

Because each id value in an MXML file is unique, all objects in a file are part of the same flat namespace. You do not qualify an object by referencing its parent with dot notation, as in myVGroup.myText.text.

For more information, see "Referring to components" on page 36.

Using XML namespaces

The xmlns property in an MXML tag specifies an XML namespace. To use the default namespace, specify no prefix. Typically, you specify a tag prefix and a namespace.

For example, the xmlns properties in the following <s:Application> tag indicates that tags corresponding to the Spark component set use the prefix *s:*.

```
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
```

Flex defines the following Universal Resource Identifiers (URI) for the Flex namespaces:

- xmlns:fx="http://ns.adobe.com/mxml/2009"

  The MXML language namespace URI. This namespace includes the top-level ActionScript language elements, such as Object, Number, Boolean, and Array. For a complete list of the top-level elements, see the Top Level package in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

  This namespace also includes the tags built in to the MXML compiler, such as <fx:Script>, <fx:Declarations>, and <fx:Style> tags. For a list of the compiler elements, see the MXML Only Tags appendix in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

  This namespace does not include the MX or Spark component sets.

  The complete list of top-level ActionScript language elements included in this namespace is defined by the frameworks\mxml-2009-manifest.xml manifest file in your Flex SDK installation directory. Note that this file does not list the MXML compiler tags because they are built in to the MXML compiler.

- xmlns:mx="library://ns.adobe.com/flex/mx"

  The MX component set namespace URI. This namespace includes all of the components in the Flex mx.* packages, the Flex charting components, and the Flex data visualization components.

  The complete list of elements included in this namespace is defined by the frameworks\mx-manifest.xml manifest file in your Flex SDK installation directory.

- xmlns:s="library://ns.adobe.com/flex/spark"

  The Spark component set namespace URI. This namespace includes all of the components in the Flex spark.* packages and the text framework classes in the flashx.* packages.

  This namespace includes the RPC classes for the WebService, HTTPService, and RemoteObject components and additional classes to support the RPC components. These classes are included in the mx: namespace, but are provided as a convenience so that you can also reference them by using the s: namespace.

  This namespace also includes several graphics, effect, and state classes from the mx.* packages. These classes are included in the mx: namespace, but are provided as a convenience so that you can also reference them by using the s: namespace.

  The complete list of elements included in this namespace is defined by the frameworks\spark-manifest.xml manifest file in your Flex SDK installation directory.

  The following table lists the classes from the mx.* packages included in this namespace:

Category	Class
RPC classes	mx.messaging.channels.AMFChannel mx.rpc.CallResponder mx.messaging.ChannelSet mx.messaging.Consumer mx.messaging.channels.HTTPChannel mx.rpc.http.mxml.HTTPService mx.messaging.Producer mx.rpc.remoting.mxml.RemoteObject mx.rpc.remoting.mxml.Operation mx.messaging.channels.RTMPChannel mx.messaging.channels.SecureAMFChannel mx.messaging.channels.SecureStreamingAMFChannel mx.messaging.channels.SecureHTTPChannel mx.messaging.channels.SecureStreamingHTTPChannel mx.messaging.channels.SecureRTMPChannel mx.messaging.channels.StreamingAMFChannel mx.messaging.channels.StreamingHTTPChannel mx.rpc.soap.mxml.WebService mx.rpc.soap.mxml.Operation mx.data.mxml.DataService
Graphics classes	mx.graphics.BitmapFill mx.geom.CompoundTransform mx.graphics.GradientEntry mx.graphics.LinearGradient mx.graphics.LinearGradientStroke mx.graphics.RadialGradient mx.graphics.RadialGradientStroke mx.graphics.SolidColor mx.graphics.SolidColorStroke mx.graphics.Stroke mx.geom.Transform

Category	Class
Effect classes	mx.effects.Parallel mx.effects.Sequence mx.states.Transition mx.effects.Wait
States classes	mx.states.State mx.states.AddItems
Component classes	mx.controls.Spacer mx.controls.SWFLoader

XML namespaces give you the ability to use classes in custom packages that are not in the Flex namespaces. The following example shows an application that contains a custom component called CustomBox. The namespace value `myComponents.boxes.*` indicates that an MXML component called CustomBox is in the myComponents/boxes directory.

```
<?xml version="1.0"?>
<!-- mxml/XMLNamespaces.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.boxes.*">

    <s:Panel title="My Application"
        height="150">
        <MyComps:CustomBox/>
    </s:Panel>
</s:Application>
```

The myComponents/boxes directory can be a subdirectory of the directory that contains the application file, or it can be a subdirectory of one of the ActionScript source path directories assigned in the flex-config.xml file. If copies of the same file exist in both places, Flex uses the file in the application file directory. The prefix name is arbitrary, but it must be used as declared.

When using a component contained in a SWC file, the package name and the namespace must match, even though the SWC file is in the same directory as the MXML file that uses it. A SWC file is an archive file for Flex components. SWC files make it easy to exchange components among Flex developers. You exchange only a single file, rather than the MXML or ActionScript files and images, along with other resource files. Also, the SWF file inside a SWC file is compiled, which means that the source code is obfuscated from casual view.

For more information on SWC files, see “Flex compilers” on page 2164.

Using MXML to trigger run-time code

Flex applications are driven by run-time events, such as when a user selects a Button control. You can specify *event listeners*, which consist of code for handling run-time events, in the event properties of MXML tags. For example, the `<s:Button>` tag has a `click` event property in which you can specify ActionScript code that executes when the Button control is clicked at run time. You can specify simple event listener code directly in event properties. To use more complex code, you can specify the name of an ActionScript function defined in an `<fx:Script>` tag.

The following example shows an application that contains a Button control and a TextArea control. The `click` property of the Button control contains a simple event listener that sets the value of the TextArea control’s `text` property to the text `Hello World`.

```
<?xml version="1.0"?>
<!-- mxml/TriggerCodeExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Panel title="My Application">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:TextArea id="textarea1"/>
        <s:Button label="Submit"
            click="textarea1.text='Hello World';"/>
    </s:Panel>
</s:Application>
```

The following example shows the code for a version of the application in which the event listener is contained in an ActionScript function in an `<fx:Script>` tag:

```
<?xml version="1.0"?>
<!-- mxml/TriggerCodeExample2.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            private function hello():void {
                textarea1.text="Hello World";
            }
        ]]>
    </fx:Script>

    <s:Panel title="My Application">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:TextArea id="textarea1"/>
        <s:Button label="Submit"
            click="hello();"/>
    </s:Panel>
</s:Application>
```

For more information about using ActionScript with MXML, see “Using ActionScript” on page 32.

Binding data between components

Flex provides simple syntax for binding the properties of components to each other. In the following example, the value inside the curly braces ({ }) binds the `text` property of a TextArea control to the `text` property of a TextInput control. When the application initializes, both controls display the text `Hello`. When the user clicks the Button control, both controls display the text `Goodbye`.

```
<?xml version="1.0"?>
<!-- mxml/BindingExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Panel title="My Application">
        <s:VGroup left="10" right="10" top="10" bottom="10">
            <s:Label text="Enter Text:"/>
            <s:TextInput id="textinput1"
                text="Hello"/>
            <s:Label text="Bind Text to the TextArea control:"/>
            <s:TextArea id="textarea1"
                text="{textinput1.text}"/>
            <s:Button label="Submit"
                click="textinput1.text='Goodbye';"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

As an alternative to the curly braces ({ }) syntax, you can use the `<fx:Binding>` tag, in which you specify the source and destination of a binding. For more information about data binding, see “Data binding” on page 299.

Using RPC services

Remote-procedure-call (RPC) services let your application interact with remote servers to provide data to your applications, or for your application to send data to a server.

Flex is designed to interact with several types of RPC services that provide access to local and remote server-side logic. For example, a Flex application can connect to a web service that uses the Simple Object Access Protocol (SOAP), a Java object residing on the same application server as Flex using AMF, or an HTTP URL that returns XML.

The MXML components that provide data access are called RPC components. MXML includes the following types of RPC components:

- WebService provides access to SOAP-based web services.
- HTTPService provides access to HTTP URLs that return data.
- RemoteObject provides access to Java objects using the AMF protocol (Adobe LiveCycle Data Services ES only).

In MXML, define the RPC components in an `<fx:Declarations>` tag. You use the `<fx:Declarations>` tag to declare non-visual components an MXML file.

The following example shows an application that calls a web service that provides weather information, and displays the current temperature for a given ZIP code. The application binds the ZIP code that a user enters in a TextInput control to a web service input parameter. It binds the current temperature value contained in the web service result to a TextArea control.

```
<?xml version="1.0"?>
<!-- mxml/RPCExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <!-- Define the web service connection
            (the specified WSDL URL is not functional). -->
        <s:WebService id="WeatherService"
            wsdl="http:/example.com/ws/WeatherService?wsdl"
            useProxy="false">
            <!-- Bind the value of the ZIP code entered in the TextInput control
                to the ZipCode parameter of the GetWeather operation. -->
            <s:operation name="GetWeather">
                <s:request>
                    <ZipCode>{zip.text}</ZipCode>
                </s:request>
            </s:operation>
        </s:WebService>
    </fx:Declarations>

    <s:Panel title="My Application">
        <s:VGroup left="10" right="10" top="10" bottom="10">
            <!-- Provide a ZIP code in a TextInput control. -->
            <s:TextInput id="zip" width="200" text="Zipcode please?"/>
            <!-- Call the web service operation with a Button click. -->
            <s:Button width="60" label="Get Weather"
                click="WeatherService.GetWeather.send();"/>
            <!-- Display the location for the specified ZIP code. -->
            <s:Label text="Location:"/>
            <s:TextArea text="{WeatherService.GetWeather.lastResult.Location}"/>
            <!-- Display the current temperature for the specified ZIP code. -->
            <s:Label text="Temperature:"/>
            <s:TextArea
                text="{WeatherService.GetWeather.lastResult.CurrentTemp}"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

For more information about using RPC services, see Accessing Server-Side Data with Flex.

Storing data in a data model

You can use a data model to store application-specific data. A *data model* is an ActionScript object that provides properties for storing data, and optionally contains methods for additional functionality. Data models provide a way to store data in the Flex application before it is sent to the server, or to store data sent from the server before using it in the application.

You can declare a simple data model that does not require methods in an `<fx:Model>`, `<fx:XML>`, or `<fx:XMLList>` tag. In MXML, define a data model in an `<fx:Declarations>` tag. You use the `<fx:Declarations>` tag to declare non-visual components an MXML file.

The following example shows an application that contains TextInput controls for entering personal contact information and a data model, represented by the `<fx:Model>` tag, for storing the contact information:

```
<?xml version="1.0"?>
<!-- mxml/StoringData.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <!-- A data model called "contact" stores contact information.
            The text property of each TextInput control shown above
            is passed to a field of the data model by using data binding. -->
        <fx:Model id="contact">
            <info>
                <homePhone>{homePhoneInput.text}</homePhone>
                <cellPhone>{cellPhoneInput.text}</cellPhone>
                <email>{emailInput.text}</email>
            </info>
        </fx:Model>
    </fx:Declarations>
    <s:Panel title="My Application">
        <s:VGroup left="10" right="10" top="10" bottom="10">
            <!-- The user enters contact information in TextInput controls. -->
            <s:TextInput id="homePhoneInput"
                text="This isn't a valid phone number."/>
            <s:TextInput id="cellPhoneInput"
                text="(999)999-999"/>
            <s:TextInput id="emailInput"
                text="me@somewhere.net"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

This example uses data binding in the model definition to automatically copy data from the UI controls to the data model.

Validating data

Flex includes a set of standard validator components for data such as phone numbers, social security numbers, and ZIP codes. You can also create your own custom validator.

In MXML, define validators in an `<fx:Declarations>` tag. You use the `<fx:Declarations>` tag to declare non-visual components an MXML file.

The following example uses validator components for validating that the expected type of data is entered in the TextInput fields. In this example, you validate a phone number by using the PhoneNumberValidator class and an e-mail address by using the EmailValidator class. Validation is triggered automatically when the user edits a TextInput control. If validation fails, the user receives immediate visual feedback.

```
<?xml version="1.0"?>
<!-- mxml/ValidatingExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <!-- Validator components validate data entered into the TextInput controls. -->
        <mx:PhoneNumberValidator id="pnV"
            source="{homePhoneInput}" property="text"/>
        <mx:EmailValidator id="emV"
            source="{emailInput}" property="text" />
    </fx:Declarations>
    <s:Panel title="My Application">
        <s:VGroup left="10" right="10" top="10" bottom="10">
            <s:Label text="Enter phone number:"/>
            <s:TextInput id="homePhoneInput"/>
            <s:Label text="Enter email address:"/>
            <s:TextInput id="emailInput"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

A component with a validation failure displays a red border. If the component has focus, it also displays a validation error message. Set the component to a valid value to remove the error indication.

For more information about using data models, see “Storing data” on page 889. For more information on validators, see “Validating Data” on page 1964.

Formatting data

Formatter components are ActionScript components that perform a one-way conversion of raw data to a formatted string. They are triggered just before data is displayed in a text field. Flex includes a set of standard formatters. You can also create your own formatters.

In MXML, define formatters in an `<fx:Declarations>` tag. You use the `<fx:Declarations>` tag to declare non-visual components an MXML file.

The following example shows an application that uses the standard ZipCodeFormatter component to format the value of a variable:

```
<?xml version="1.0"?>
<!-- mxml/FormatterExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            [Bindable]
            private var storedZipCode:Number=123456789;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Declare a ZipCodeFormatter and define parameters. -->
        <mx:ZipCodeFormatter id="ZipCodeDisplay" formatString="#####-####"/>
    </fx:Declarations>
    <s:Panel title="My Application">
        <!-- Trigger the formatter while populating a string with data. -->
        <s:TextInput text="{ZipCodeDisplay.format(storedZipCode)}"/>
    </s:Panel>
</s:Application>
```

For more information about formatter components, see “Formatting Data” on page 2004.

Using Cascading Style Sheets (CSS)

You can use style sheets based on the CSS standard to declare styles to Flex components. The MXML `<fx:Style>` tag contains inline style definitions or a reference to an external file that contains style definitions.

The `<fx:Style>` tag must be an immediate child of the root tag of the MXML file. You can apply styles to an individual component using a class selector, or to all components of a certain type using a type selector.

Namespace qualification is required for type selectors in the `<fx:Style>` tag. Prefix the namespace qualification with the `@namespace` tag.

The following example defines a class selector and a type selector in the `<fx:Style>` tag. Both the class selector and the type selector are applied to the Button control.

```
<?xml version="1.0"?>
<!-- mxml/CSSExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";

        /* class selector */
        .myClass {
            color: Red
        }

        /* type selector */
        s|Button {
            font-size: 18pt
        }
    </fx:Style>
    <s:Panel title="My Application">
        <s:Button styleName="myClass" label="This is red 18 point text."/>
    </s:Panel>
</s:Application>
```

A class selector in a style definition, defined as a label preceded by a period, defines a new named style, such as `myClass` in the preceding example. After you define it, you can apply the style to any component by using the `styleName` property. In the preceding example, you apply the style to the Button control to set the font color to red.

A type selector applies a style to all instances of a particular component type. In the preceding example, you set the font size for all Spark Button controls to 18 points.

For more information about using Cascading Style Sheets, see “Styles and themes” on page 1492.

Using skins

Skinning is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of bitmap images, SWF files, or class files that contain drawing methods that define vector images. Skins can define the entire appearance, or only a part of the appearance, of a component in various states.

One of the big differences between Spark and MX components is that Spark components rely on the component skin to define its layout and appearance. When working with Spark components, you often define a custom skin to modify the component appearance.

MX components use a combination of CSS styles and skins to control their appearance. With MX components, you can use styles to modify much of the appearance of the component without having to define a custom skin.

For more information about using Spark skins, see “Spark Skinning” on page 1602. For more information about using MX skins, see “Skinning MX components” on page 1655.

Using effects

An *effect* is a change to a component that occurs over a brief period of time. Examples of effects are fading, resizing, and moving a component. In MXML, you apply effects as properties of a control or container. Flex provides a set of built-in effects with default properties.

In MXML, define effects in an `<fx:Declarations>` tag. You use the `<fx:Declarations>` tag to declare non-visual components an MXML file.

The following example shows an application that contains a Button control with its `click` property set to use the Resize effect when the user moves the mouse over it:

```
<?xml version="1.0"?>
<!-- behaviors\TargetProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Resize id="myResize"
            heightBy="25"
            widthBy="50"
            target="{myButton}"/>
    </fx:Declarations>

    <s:Button id="myButton"
        label="Resize target"
        click="myResize.end();myResize.play();"/>
</s:Application>
```

For more information about effects, see “Introduction to effects” on page 1784.

Defining custom MXML components

Custom MXML components are MXML files that you create and use as custom MXML tags in other MXML files. They encapsulate and extend the functionality of existing Flex components. Just like MXML application files, MXML component files can contain a mix of MXML tags and ActionScript code. The name of the MXML file becomes the class name with which you refer to the component in another MXML file.

The following example shows a custom ComboBox control that is prepopulated with list items:

```
<?xml version="1.0"?>
<!-- mxml/myComponents/boxes/MyComboBox.mxml -->
<s:VGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

     <s:ComboBox>
        <s:dataProvider>
            <s:ArrayCollection>
                <fx:String>Dogs</fx:String>
                <fx:String>Cats</fx:String>
                <fx:String>Mice</fx:String>
            </s:ArrayCollection>
        </s:dataProvider>
    </s:ComboBox>
</s:VGroup>
```

The following example shows an application that uses the MyComboBox component as a custom tag. The value `myComponents.boxes.*` assigns the `MyComps` namespace to the myComponents/boxes sub-directory. To run this example, store the MyComboBox.mxml file in that sub-directory.

```
<?xml version="1.0"?>
<!-- mxml/CustomMXMLComponent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.boxes.*">
    <s:Panel title="My Application"
        height="150">
        <MyComps:MyComboBox/>
    </s:Panel>
</s:Application>
```

For more information about MXML components, see “Simple MXML components” on page 2399. You can also define custom Flex components in ActionScript. For more information, see “Create simple visual components in ActionScript” on page 2433.

MXML syntax

MXML is an XML language that you use to lay out user-interface components for Adobe® Flex® applications.

Basic MXML syntax

Most MXML tags correspond to ActionScript 3.0 classes or properties of classes. Flex parses MXML tags and compiles a SWF file that contains the corresponding ActionScript objects.

ActionScript 3.0 uses syntax based on the ECMAScript edition 4 draft language specification. ActionScript 3.0 includes the following features:

- Formal class definition syntax
- Formal packages structure
- Typing of variables, parameters, and return values (compile-time only)
- Implicit getters and setters that use the `get` and `set` keywords
- Inheritance
- Public and private members
- Static members
- Cast operator

For more information about ActionScript 3.0, see “Using ActionScript” on page 32.

Naming MXML files

MXML filenames must adhere to the following naming conventions:

- Filenames must be valid ActionScript identifiers, which means they must start with a letter or underscore character (_), and they can only contain letters, numbers, and underscore characters after that.
- Filenames must not be the same as ActionScript class names, component `id` values, or the word *application*. Do not use filenames that match the names of MXML tags that are in the `fx:`, `s:`, or `mx:` namespace.
- Filenames must end with a lowercase .mxml file extension.

Using tags that represent ActionScript classes

An MXML tag that corresponds to an ActionScript class uses the same naming conventions as the ActionScript class. Class names begin with a capital letter, and capital letters separate the words in class names. For example, when a tag corresponds to an ActionScript class, its properties correspond to the properties and events of that class.

Setting component properties

In MXML, a component property uses the same naming conventions as the corresponding ActionScript property. A property name begins with a lowercase letter, and capital letters separate words in the property names.

You can set most component properties as tag attributes, in the form:

```
<s:Label width="50" height="25" text="Hello World"/>
```

You can set all component properties as child tags, in the form:

```
<s:Label>
    <s:width>50</s:width>
    <s:height>25</s:height>
    <s:text>Hello World</s:text>
</s:Label>
```

You often use child tags when setting the value of a property to a complex Object because it is not possible to specify a complex Object as the value of tag attribute. In the following example, you use child tags to set the data provider of a Spark List control to an ArrayCollection object:

```
<s:List>
    <s:dataProvider>
        <s:ArrayCollection>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
            <fx:String>AR</fx:String>
        </s:ArrayCollection>
    </s:dataProvider>
</s:List>
```

The one restriction on setting properties that use child tags is that the namespace prefix of a child tag, `s:` in the previous example, must match the namespace prefix of the component tag. Note that the *value* of the property in the previous example does not have to use the same namespace as the property tag.

Each of a component's properties is one of the following types:

- Scalar properties, such as a number or string
- Array of scalar values, such as an array of numbers or strings
- ActionScript object
- Array of ActionScript objects
- ActionScript properties
- XML data

Adobe recommends that you assign scalar values using tag attributes, and that you assign complex types, such as ActionScript objects, by using child tags.

Setting scalar properties

You usually specify the value of a scalar property as a property of a component tag, as the following example shows:

```
<s:Label width="50" height="25" text="Hello World"/>
```

Setting properties using constants

The valid values of many component properties are defined by static constants, where these static constants are defined in an ActionScript class. In MXML, you can either use the static constant to set the property value, or use the value of the static constant, as the following example shows:

```
<!-- Set the property using the static constant. -->
<s:Wipe direction="{spark.effects.WipeDirection.LEFT}">
    ...
</s:Wipe>

<!-- Set the property using the value of the static constant. -->
<s:Wipe direction="left">
    ...
</s:Wipe>
```

The Wipe effect defines a property named `direction` that defines the direction of the wipe effect. In this example, you explicitly set the `direction` property to cause the wipe effect to move left.

In the first example, you set the `direction` property using a static constant named `LEFT`, which is defined in the spark.effects.WipeDirection class. In MXML, you must use data binding syntax when setting a property value to a static constant. The advantage of using the static constant is that the Flex compiler recognizes incorrect property values, and issues an error message at compile time.

Alternatively, you can set the value of the `direction` property to the value of the static constant. The value of the `LEFT` static constant is `"left"`. When you use the value of the static constant to set the property value, the Flex compiler cannot determine if you used an unsupported value. If you incorrectly set the property, you will not know until you get a run-time error.

In ActionScript, you should use static constants to set property values whenevera possible, as the following example shows:

```
var myWipe:Wipe = new Wipe();
myWipe.direction=spark.effects.WipeDirection.LEFT;
```

Setting the default property

Many Flex components define a single default property. The *default property* is the MXML tag that is implicit for content inside of the MXML tag if you do not explicitly specify a property. For example, consider the following MXML tag definition:

```
<s:SomeTag>
    anything here
</s:SomeTag>
```

If this tag defines a default property named `default_property`, the preceding tag definition is equivalent to the following code:

```
<s:SomeTag>
    <s:default_property>
        anything here
    </s:default_property>
</s:SomeTag>
```

It is also equivalent to the following code:

```
<s:SomeTag default_property="anything here"/>
```

The default property provides a shorthand mechanism for setting a single property. For a Spark List, the default property is the `dataProvider` property. Therefore, the two List definitions in the following code are equivalent:

```
<?xml version="1.0"?>
<!-- mxml\DefProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <!-- Omit the default property. -->
    <s:List>
        <s:ArrayCollection>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
            <fx:String>AR</fx:String>
        </s:ArrayCollection>
    </s:List>

    <!-- Explicitly speficy the default property. -->
    <s:List>
        <s:dataProvider>
            <s:ArrayCollection>
                <fx:String>AK</fx:String>
                <fx:String>AL</fx:String>
                <fx:String>AR</fx:String>
            </s:ArrayCollection>
        </s:dataProvider>
    </s:List>
</s:Application>
```

Not all Flex components define a default property. To determine the default property for each component, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

You can also define a default property when you create a custom component. For more information, see “Metadata tags in custom components” on page 2376.

Escaping characters using the backslash character

When setting a property value in MXML, you can escape a reserved character by prefixing it with the backslash character (\), as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml\EscapeChar.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Label text="\{\}"/>
</s:Application>
```

In this example, you want to use literal curly brace characters ({ }) in a text string. But Flex uses curly braces to indicate a data binding operation. Therefore, you prefix each curly brace with the backslash character to cause the MXML compiler to interpret them as literal characters.

Setting String properties using the backslash character

The MXML compiler automatically escapes the backslash character in MXML when the character is part of the value specified for a property of type String. Therefore, it always converts `"\"` to `"\\"`.

This is necessary because the ActionScript compiler recognizes `"\\"` as the character sequence for a literal "\" character, and strips out the leading backslash when it initializes the property value.

***Note:** Do not use the backslash character (\) as a separator in the path to an application asset. You should always use a forward slash character (/) as the separator.*

Including a newline character in a String value

For properties of type String, you can insert a newline character in the String in several ways:

- By using data binding with the '\n' characters in your String value in MXML
- By inserting the `&#13;` code in your String value in MXML
- By inserting `"\n"` in an ActionScript String variable used to initialize the MXML property

To use data binding, wrap the newline character in curly brace characters ({ }), as the following example shows:

```
<s:TextArea width="100%" text="Display{'\n'}Content"/>
```

To use the `&#13;` code to insert a newline character, include that code in the property value in MXML, as the following example shows:

```
<s:TextArea width="100%" text="Display&#13;Content"/>
```

To use an ActionScript String variable to insert a newline character, create an ActionScript variable, and then use data binding to set the property in MXML, as the following example shows:

```
<fx:Script>
    <![CDATA[
        [Bindable]
        public var myText:String = "Display" + "\n" + "Content";
    ]]>
</fx:Script>

<s:TextArea width="100%" text="{myText}"/>
```

In this example, you set the `text` property of the TextArea control to a value that includes a newline character.

Notice that this example includes the `[Bindable]` metadata tag before the property definition. This metadata tag specifies that the `myText` property can be used as the source of a data binding expression. Data binding automatically copies the value of a `source` property of one object to the destination property of another object at run time when the `source` property changes.

If you omit this metadata tag, the compiler issues a warning message specifying that the property cannot be used as the source for data binding. For more information, see "Data binding" on page 299.

Mixing content types

Flex supports the mixing of content for attribute values. The attribute must correspond to a property of type Object or Array. Mixed content consists of non white space character data and MXML tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml\DefPropMixed.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:RichText  fontFamily="Verdana" fontWeight="bold">
        <s:content>
            <fx:String>Hello World!</fx:String><s:br/>
            <fx:String>Hello Universe</fx:String><s:br/>
            Hello Flex!
        </s:content>
    </s:RichText>
</s:Application>
```

In this example, the `RichText.content` property is of type Object. The property value contains values defined by `<fx:String>` tag, and by character data ("Hello Flex!"). Character data is almost always converted to a value of type String. However, an Array property may be defined to support an explicit data type for its values. In that case, the compiler converts the character data to the appropriate data type.

Setting Arrays of scalar values

When a class has a property that takes an Array as its value, you can represent the property in MXML using child tags. The component in the following example has a `children` property that contains an Array of numbers:

```
<mynamespace:MyComponent>
    <mynamespace:children>
        <fx:Array>
            <fx:Number>94062</fx:Number>
            <fx:Number>14850</fx:Number>
            <fx:Number>53402</fx:Number>
        </fx:Array>
    </mynamespace:children>
</mynamespace:MyComponent>
```

The `<fx:Array>` and `</fx:Array>` tags around the Array elements are optional. Therefore, you can also write this example as the following code shows:

```
<mynamespace:MyComponent>
    <mynamespace:children>
        <fx:Number>94062</fx:Number>
        <fx:Number>14850</fx:Number>
        <fx:Number>53402</fx:Number>
    </mynamespace:children>
</mynamespace:MyComponent>
```

In this example, since the data type of the `children` property is defined as Array, Flex automatically converts the three number definitions into a three-element array.

Component developers may have specified additional information within the component definition that defines the data type of the Array elements. For example, if the developer specified that the `dataProvider` property supports only String elements, this example would cause a compiler error because you specified numbers to it. The *ActionScript 3.0 Reference for the Adobe Flash Platform* documents the Array properties that define a required data type for the Array elements.

Setting Object properties

When a component has a property that takes an object as its value, you can represent the property in MXML using a child tag with tag attributes, as the following example shows:

```
<mynamespace:MyComponent>
    <mynamespace:nameOfProperty>
        <mynamespace:typeOfObject prop1="val1" prop2="val2"/>
    </mynamespace:nameOfProperty>
</mynamespace:MyComponent>
```

The following example shows an ActionScript class that defines an Address object. This object is used as a property of the PurchaseOrder component in the next example.

```
class Address {
    public var name:String;
    public var street:String;
    public var city:String;
    public var state:String;
    public var zip:Number;
}
```

The following example shows an ActionScript class that defines a PurchaseOrder component that has a property type of Address:

```
import example.Address;

class PurchaseOrder {
    public var shippingAddress:Address;
    public var quantity:Number;
...
}
```

In MXML, you define the PurchaseOrder component as the following example shows:

```
<mynamespace:PurchaseOrder quantity="3" xmlns:e="example">
    <mynamespace:shippingAddress>
        <mynamespace:Address name="Fred" street="123 Elm St."/>
    </mynamespace:shippingAddress>
</mynamespace:PurchaseOrder>
```

If the value of the `shippingAddress` property is a subclass of Address (such as DomesticAddress), you can declare the property value, as the following example shows:

```
<mynamespace:PurchaseOrder quantity="3" xmlns:e="example">
    <mynamespace:shippingAddress>
        <mynamespace:DomesticAddress name="Fred" street="123 Elm St."/>
    </mynamespace:shippingAddress>
</mynamespace:PurchaseOrder>
```

If the property is explicitly typed as Object like the `value` property in the following example, you can specify an anonymous object using the `<fx:Object>` tag.

```
class ObjectHolder {
    public var value:Object
}
```

The following example shows how you specify an anonymous object as the value of the `value` property:

```
<mynamespace:ObjectHolder>
    <mynamespace:value>
        <fx:Object foo='bar'/>
    </mynamespace:value>
</mynamespace:ObjectHolder>
```

Populating an Object with an Array

When a component has a property of type Object that takes an Array as its value, you can represent the property in MXML using child tags, as the following example shows:

```
<mynamespace:MyComponent>
    <mynamespace:nameOfObjectProperty>
        <fx:Array>
            <fx:Number>94062</fx:Number>
            <fx:Number>14850</fx:Number>
            <fx:Number>53402</fx:Number>
        </fx:Array>
    </mynamespace:nameOfObjectProperty>
</mynamespace:MyComponent>
```

In this example, you initialize the Object property to a three-element array of numbers.

As described in the section "Setting Arrays of scalar values" on page 26, the `<fx:Array>` tag and the `</fx:Array>` tag around the Array elements are optional and may be omitted, as the following example shows:

```
<mynamespace:MyComponent>
    <mynamespace:nameOfObjectProperty>
        <fx:Number>94062</fx:Number>
        <fx:Number>14850</fx:Number>
        <fx:Number>53402</fx:Number>
    </mynamespace:nameOfObjectProperty>
</mynamespace:MyComponent>
```

The only exception to this rule is when you specify a single Array element for the Object property. In that case, Flex does not create an Object containing a single-element array, but instead creates an object and sets it to the specified value. This is a difference between the following two lines:

```
object=[element] // Object containing a one-element array
object=element // object equals value
```

If you want to create a single-element array, include the `<fx:Array>` and `</fx:Array>` tags around the array element, as the following example shows:

```
<mynamespace:MyComponent>
    <mynamespace:nameOfObjectProperty>
        <fx:Array>
            <fx:Number>94062</fx:Number>
        </fx:Array>
    </mynamespace:nameOfObjectProperty>
</mynamespace:MyComponent>
```

Populating Arrays of objects

When a component has a property that takes an Array of objects as its value, you can represent the property in MXML using child tags, as the following example shows:

```
<mynamespace:MyComponent>
    <mynamespace:nameOfProperty>
        <fx:Array>
            <mynamespace:objectType prop1="val1" prop2="val2"/>
            <mynamespace:objectType prop1="val1" prop2="val2"/>
            <mynamespace:objectType prop1="val1" prop2="val2"/>
        </fx:Array>
    </mynamespace:nameOfProperty>
</mynamespace:MyComponent>
```

The component in the following example contains an Array of ListItem objects. Each ListItem object has properties named `label` and `data`.

```
<mynamespace:MyComponent>
    <mynamespace:dataProvider>
        <fx:Array>
            <mynamespace:ListItem label="One" data="1"/>
            <mynamespace:ListItem label="Two" data="2"/>
        </fx:Array>
    </mynamespace:dataProvider>
</mynamespace:MyComponent>
```

The following example shows how you specify an anonymous object as the value of the `dataProvider` property:

```
<mynamespace:MyComponent>
    <mynamespace:dataProvider>
        <fx:Array>
            <fx:Object label="One" data="1"/>
            <fx:Object label="Two" data="2"/>
        </fx:Array>
    </mynamespace:dataProvider>
</mynamespace:MyComponent>
```

As described in the section “Setting Arrays of scalar values” on page 26, the `<fx:Array>` tag and the `</fx:Array>` tag around the Array elements are optional and may be omitted, as the following example shows:

```
<mynamespace:MyComponent>
    <mynamespace:dataProvider>
        <fx:Object label="One" data="1"/>
        <fx:Object label="Two" data="2"/>
    </mynamespace:dataProvider>
</mynamespace:MyComponent>
```

Populating a Vector

The Vector class lets you access and manipulate a vector. A Vector is an array whose elements all have the same data type. The data type of a Vector's elements is known as the Vector's *base* type. The base type can be any class, including built in classes and custom classes. The base type is specified when declaring a Vector variable as well as when creating an instance by calling the class constructor.

In MXML, define an instance of a Vector class in a `<fx:Declarations>` block, as the following example shows:

```
<fx:Declarations>
    <fx:Vector type="String">
        <fx:String>one</fx:String>
        <fx:String>two</fx:String>
        <fx:String>three</fx:String>
    </fx:Vector>

    <fx:Vector type="Vector.&lt;String&gt;">
        <fx:Vector type="String">
            <fx:String>one</fx:String>
            <fx:String>two</fx:String>
            <fx:String>three</fx:String>
        </fx:Vector>
    </fx:Vector>
</fx:Declarations>
```

The first example defines a Vector with a base type of String. The second example defines a Vector of Vectors of type String. Notice that you use the HTML escape characters `&lt;` and `&gt;`, instead of < and >, in the nested Vector. This syntax is necessary to conform to XML syntax rules.

Setting properties that contain XML data

If a component contains a property that takes XML data, the value of the property is an XML fragment to which you can apply a namespace. In the following example, the `value` property of the MyComponent object is XML data:

```
<mynamespace:MyComponent>
    <mynamespace:value xmlns:a="http://www.example.com/myschema">
        <fx:XML>
            <a:purchaseorder>
                <a:billingaddress>
                ...
                </a:billingaddress>
                ...
            </a:purchaseorder>
        </fx:XML>
    </mynamespace:value>
</mynamespace:MyComponent>
```

Setting style properties in MXML

A style property of an MXML tag differs from other properties because it corresponds to an ActionScript style, rather than to a property of an ActionScript class. You set these properties in ActionScript using the `setStyle(`*stylename*`,` *value*`)` method rather than `object.property=value` notation.

For example, you can set the `fontFamily` style property in MXML, as the following code shows:

```
<s:TextArea id="myText" text="hello world" fontFamily="Tahoma"/>
```

This MXML code is equivalent to the following ActionScript code:

```
myText.setStyle("fontFamily", "Tahoma");
```

You define style properties in custom ActionScript classes by using the `[Style]` metadata tags, rather than defining them as ActionScript variables or setter/getter methods. For more information, see “Metadata tags in custom components” on page 2376.

Setting event properties in MXML

An event property of an MXML tag lets you specify the event listener function for an event. This property corresponds to setting the event listener in ActionScript using the `addEventListener()` method.

For example, you can set the `creationComplete` event property in MXML, as the following code shows:

```
<s:TextArea id="myText" creationComplete="creationCompleteHandler();"/>
```

This MXML code is equivalent to the following ActionScript code:

```
myText.addEventListener("creationComplete", creationCompleteHandler);
```

You define event properties in custom ActionScript classes by using the `[Event]` metadata tags, rather than defining them as ActionScript variables or setter/getter methods. For more information, see “Metadata tags in custom components” on page 2376.

Specifying a URL value

Some MXML tags, such as the `<fx:Script>` tag, have a property that takes a URL of an external file as a value. For example, you can use the `source` property in an `<fx:Script>` tag to reference an external ActionScript file instead of typing ActionScript directly in the body of the `<fx:Script>` tag.

Note: *You specify a script in the* `source` *property of an* `<fx:Script>` *tag. You do not specify ActionScript classes in the* `source` *property. For information on using ActionScript classes, see "Creating ActionScript components" on page 49.*

MXML supports the following types of URLs:

- Absolute, as in the following example:

  ```
  <fx:Style source="http://www.somesite.com/mystyles.css">
  ```

- A path used at compile time that is relative to the application, as in the following example:

  ```
  <fx:Script source="/myscript.as"/>
  ```

- Relative to the current file location, as in the following example:

  ```
  <fx:Script source="../myscript.as"/>
  ```

- A path used at run time that is relative to the context root of the Java web application in which a Flex application is running. For example:

  ```
  <mx:HTTPService url="@ContextRoot()/directory/myfile.xml"/>
  ```

  You can only use the `@ContextRoot()` token if you are using the web-tier compiler or if you set the value of the `context-root` compiler argument

Specifying a RegExp value

For a property of type RegExp, you can specify its value in MXML using the following format:

```
"/pattern/flags"
```

pattern Specifies the regular expression within the two slashes. Both slashes are required.

flags (Optional) Specifies any flags for the regular expression.

For example, the `regExpression` property of an MXML component is of type RegExp. Therefore, you can set its value as the following example shows:

```
<mynamespace:MyComponent regExpression="/\Wcat/gi"/>
```

Or set it using child tags, as the following example shows:

```
<mynamespace:MyComponent>
    <mynamespace:regExpression>/\Wcat/gi</mynamespace:regExpression>
</mynamespace:MyComponent>
```

The *flags* portion of the regular expression is optional, so you can also specify it as the following example shows:

```
<mynamespace:MyComponent regExpression="/\Wcat/"/>
```

Using compiler tags

Compiler tags are tags that do not directly correspond to ActionScript objects or properties. They include the following:

- `<fx:Binding>`
- `<fx:Component>`
- `<fx:Declarations>`

- `<fx:Definition>`
- `<fx:DesignLayer>`
- `<fx:Library>`
- `<fx:Metadata>`
- `<fx:Model>`
- `<fx:Private>`
- `<fx:Reparent>`
- `<fx:Script>`
- `<fx:Style>`

MXML tag rules

MXML has the following syntax requirements:

- The `id` property is not required on any tag.
- The `id` property is not allowed on the root tag.
- Boolean properties take only `true` and `false` values.
- The `<fx:Binding>` tag requires both `source` and `destination` properties.
- The `<fx:Binding>` tag cannot contain an `id` property.

Using ActionScript

Flex developers can use ActionScript to extend the functionality of their Adobe® Flex® applications. ActionScript provides flow control and object manipulation features that are not available in MXML. For a complete introduction to ActionScript and a reference for using the language, see *ActionScript 3.0 Developer's Guide* and *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Using ActionScript in applications

Flex developers can use ActionScript to implement custom behavior within their applications. You first use MXML tags to declare things like the containers, controls, effects, formatters, validators, and web services that your application requires, and to lay out its user interface. Each of these components provides the standard behavior you'd expect. For example, a button automatically highlights when you roll over it, without requiring you to write any ActionScript. But a declarative language like MXML is not appropriate for coding what you want to happen when the user clicks a button. For that, you need to use a procedural language like ActionScript, which offers executable methods, various types of storage variables, and flow control such as conditionals and loops. In a general sense, MXML implements the static aspects of your application, and ActionScript implements its dynamic aspects.

ActionScript is an object-oriented procedural programming language, based on the ECMAScript (ECMA-262) edition 4 draft language specification. You can use a variety of methods to mix ActionScript and MXML, including the following:

- Define event listeners inside MXML event attributes.
- Add script blocks using the `<fx:Script>` tag.
- Include external ActionScript files.

- Import ActionScript classes.
- Create ActionScript components.

ActionScript compilation

Although a simple application can be written in a single MXML or ActionScript (AS) file, most applications will be broken into multiple files. For example, it is common to move the `<fx:Script>` and `<fx:Style>` blocks into separate AS and CSS files that the application then includes.

It is also common for an application to import custom MXML and ActionScript components. These must be defined in other files, and MXML components may put their own `<fx:Script>` blocks into yet more AS files that they include. Components may also be imported from precompiled SWC files rather than source code. Finally, SWF files containing executable code can also be embedded in an application. The end result of all these input files is a single SWF file.

The Flex compiler transforms the main MXML file and other files it includes into a single ActionScript class. As a result, you cannot define classes or use statements outside of functions in the MXML files and the included ActionScript files.

You can reference imported ActionScript classes from your MXML application files, and those classes are added to the final SWF file. When the transformation to an ActionScript file is complete, Flex links all the ActionScript components and includes those classes in the final SWF file.

About generated ActionScript

When you write an MXML file and compile it, the Flex compiler creates a class and generates ActionScript that the class uses. MXML tags and ActionScript are used by the resulting class in several ways. This information is useful for understanding what is happening in the background of the application.

An MXML application (a file that starts with the `<s:Application>` tag) defines a subclass of the Spark Application class. Similarly, an MXML component (a file that starts with some other component's tag, such as `<s:Button>`) defines a subclass of that component.

The name of the subclass is the name of the file. The base class is the class of the top-level tag. An MXML application actually defines the following:

```
class MyApp extends Application
```

If MyButton.mxml starts with `<s:Button>`, you are actually defining the following:

```
class MyButton extends Button
```

The variable and function declarations in an `<fx:Script>` block define properties and methods of the subclass.

Setting an `id` property on a component instance within a class results in a public variable being autogenerated in the subclass that contains a reference to that component instance. For example, if the `<s:Button id="myButton"/>` tag is nested deeply inside several containers, you can still refer to it as `myButton`.

Event attributes become the bodies of autogenerated event listener methods in the subclass. For example:

```
<s:Button id="myButton" click="foo = 1; doSomething()">
```

becomes

```
private function __myButton_click(event:MouseEvent):void {
    foo = 1;
    doSomething()
}
```

The event attributes become method bodies, so they can access the other properties and methods of the subclass.

All the ActionScript anywhere in an MXML file, whether in its `<fx:Script>` block or inside tags, executes with the `this` keyword referring to an instance of the subclass.

The public properties and methods of the class are accessible by ActionScript code in other components, as long as that code “dots down” (for example, `myCheckoutAccordion.myAddressForm.firstNameTextInput.text`) or reaches up using the `parentDocument`, `parentApplication`, or `FlexGlobals.topLevelApplication` properties to specify which component the property or method exists on.

Using ActionScript in MXML event handlers

One way to use ActionScript code in an application is to include it within the MXML tag’s event handler, as the following example shows:

```
<?xml version="1.0"?>
<!-- usingas/HelloWorldAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:BasicLayout/>
    </s:layout>
    <s:Panel title="My Application" height="128" x="226" y="24">
        <s:TextArea id="textarea1" width="155" x="0" y="0"/>
        <s:Button label="Click Me"
            click="textarea1.text='Hello World';"
            width="92"
            x="31.5" y="56"/>
    </s:Panel>
</s:Application>
```

In this example, you include ActionScript code for the body of the click event handler of the Button control. The MXML compiler takes the attribute `click="..."` and generates the following event handler method:

```
public function __myButton_click(event:MouseEvent):void {
    textarea1.text='Hello World';
}
```

When the user clicks the button, this code sets the value of the TextArea control’s `text` property to the String `"Hello World."` In most cases, you do not need to look at the generated code, but it is useful to understand what happens when you write an inline event handler.

To see the generated code, set the value of the `keep-generated-actionscript` compiler option to `true`. The compiler then stores the *.as helper file in the /generated directory, which is a subdirectory of the location of the SWF file.

For more information about events, see “Events” on page 54. For more information on using the command-line compilers, see “Flex compilers” on page 2164.

Using ActionScript blocks in MXML files

You use the `<fx:Script>` tag to insert an ActionScript block in an MXML file. ActionScript blocks can contain ActionScript functions and variable declarations used in MXML applications. Code inside `<fx:Script>` tags can also declare constants (with the `const` statement) and namespaces (with `namespace`), include ActionScript files (with `include`), import declarations (with `import`), and use namespaces (with `use namespace`).

The `<fx:Script>` tag must be a child of the `<s:Application>` or other top-level component tag.

Statements and expressions are allowed only if they are wrapped in a function. In addition, you cannot define new classes or interfaces in `<fx:Script>` blocks. Instead, you must place new classes or interfaces in separate AS files and import them.

All ActionScript in the block is added to the enclosing file's class when Flex compiles the application. The following example declares a variable and sets the value of that variable inside a function:

```
<?xml version="1.0"?>
<!-- usingas/StatementSyntax.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="doSomething()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
        public var s:Boolean;

        public function doSomething():void {
            // The following statements must be inside a function.
            s = label1.visible;
            label1.text = "label1.visible = " + String(s);
        }
        ]]>
    </fx:Script>

    <s:Label id="label1"/>

</s:Application>
```

Most ActionScript statements must be inside functions in an `<fx:Script>` block. However, the following statements can be outside functions:

- `import`
- `var`
- `include`
- `const`
- `namespace`
- `use namespace`

When using an `<fx:Script>` block, you should wrap the contents in a CDATA construct. This prevents the compiler from interpreting the contents of the script block as XML, and allows the ActionScript to be properly generated. Adobe recommends that you write all your `<fx:Script>` open and close tags as the following example shows:

```
<fx:Script>
    <![CDATA[
        ...
    ]]>
</fx:Script>
```

Flex does not parse text in a CDATA construct, which means that you can use XML-parsed characters such as angle brackets (< and >) and ampersand (&). For example, the following script that includes a greater-than (>) comparison must be in a CDATA construct:

```
<?xml version="1.0"?>
<!-- usingas/UsingCDATA.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="doSomething()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        public var m:Number;
        public var n:Number;

        public function doSomething():void {
            n = 40;
            m = 42;
            label1.text = "40 < 42 = " + String(n < m);
        }
        ]]>
    </fx:Script>

    <s:Label id="label1"/>

</s:Application>
```

Accessing ActionScript documentation

The ActionScript 3.0 programming language can be used from within several development environments, including Adobe® Flash® Professional and Adobe® Flash® Builder™.

The Flex documentation includes *ActionScript 3.0 Developer's Guide*, which describes the ActionScript language. The ActionScript API reference is included as part of the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Working with components

The primary use of ActionScript in your applications is probably going to be for working with the visual controls and containers in your application. Flex provides several techniques for doing this, including referencing a Flex control in ActionScript and manipulating properties during the instantiation of controls and containers.

Referring to components

To work with a component in ActionScript, you usually define an `id` property for that component in the MXML tag. For example, the following code sets the `id` property of the Button control to the String `"myButton"`:

```
<s:Button id="myButton" label="Click Me"/>
```

This property is optional if you do not want to access the component with ActionScript.

This code causes the MXML compiler to autogenerate a public variable named `myButton` that contains a reference to that Button instance. This autogenerated variable lets you access the component instance in ActionScript. You can explicitly refer to the Button control's instance with its `id` instance reference in any ActionScript class or script block. By referring to a component's instance, you can modify its properties and call its methods.

For example, the following ActionScript block changes the value of the Button control's `label` property when the user clicks the button:

```
<?xml version="1.0"?>
<!-- usingas/ButtonExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        private function setLabel():void {
            if (myButton.label=="Click Me") {
                myButton.label = "Clicked";
            } else {
                myButton.label = "Click Me";
            }
        }
        ]]>
    </fx:Script>
    <s:Button id="myButton" label="Click Me" click="setLabel();"/>

</s:Application>
```

The IDs for all tags in an MXML component, no matter how deeply nested they are, generate public variables of the component being defined. As a result, all `id` properties must be unique within a document. This also means that if you specified an ID for a component instance, you can access that component from anywhere in the application: from functions, external class files, imported ActionScript files, or inline scripts.

You can refer to a component if it does not have an `id` property by using methods of the component's Spark container, such as the `getElementAt()` method. For MX containers, you can use the `getChildAt()` method.

You can refer to the current enclosing document or current object using the `this` keyword.

You can also get a reference to a component when you have a String that matches the name. To access an object on the application, you use the `this` keyword, followed by square brackets, with the String inside the square brackets. The result is a reference to the objects whose name matches the String.

The following example changes style properties on each Button control using a compound String to get a reference to the object:

```
<?xml version="1.0"?>
<!-- usingas/FlexComponents.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private var newFontStyle:String;
        private var newFontSize:int;
        public function changeLabel(s:String):void {
            s = "myButton" + s;
            if (this[s].getStyle("fontStyle")=="normal") {
                newFontStyle = "italic";
                newFontSize = 18;
            } else {
                newFontStyle = "normal";
                newFontSize = 10;
            }
            this[s].setStyle("fontStyle",newFontStyle);
            this[s].setStyle("fontSize",newFontSize);
        }
        ]]>
    </fx:Script>
    <s:Button id="myButton1"
        click="changeLabel('2')"
        label="Change Other Button's Styles"/>
    <s:Button id="myButton2"
        click="changeLabel('1')"
        label="Change Other Button's Styles"/>
</s:Application>
```

This technique is especially useful if you use a Repeater control or when you create objects in ActionScript and do not necessarily know the names of the objects you want to refer to prior to run time. However, when you instantiate an object in ActionScript, to add that object to the properties array, you must declare the variable as public and declare it in the class's scope, not inside a function.

The following example uses ActionScript to declare two Label controls in the application scope. During initialization, the labels are instantiated and their `text` properties are set. The example then gets a reference to the Label controls by appending the passed-in variable to the String when the user clicks the Button controls.

```
<?xml version="1.0"?>
<!-- usingas/ASLabels.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initLabels()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.controls.Label;
        public var label1:Label;
        public var label2:Label;
        // Objects must be declared in the application scope. Adds the names to
        // the application's properties array.
        public function initLabels():void {
            label1 = new Label();
            label1.text = "Change Me";
            label2 = new Label();
            label2.text = "Change Me";
            addElement(label1);
            addElement(label2);
        }
        public function changeLabel(s:String):void {
            // Create a String that matches the name of the Label control.
            s = "label" + s;
            // Get a reference to the label control using the
            // application's properties array.
            this[s].text = "Changed";
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" click="changeLabel('2')" label="Change Other Label"/>
    <s:Button id="b2" click="changeLabel('1')" label="Change Other Label"/>
</s:Application>
```

Calling component methods

You can invoke the public methods of a component instance in your application by using the following dot-notation syntax:

```
componentInstance.method([parameters]);
```

The following example invokes the `adjustThumb()` method when the user clicks the button, which invokes the public `setThumbValueAt()` method of the HSlider control:

```
<?xml version="1.0"?>
<!-- usingas/ComponentMethods.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        public function adjustThumb(s:HSlider):void {
            var randomNumber:int = (Math.floor(Math.random() * 10));
            s.setThumbValueAt(0, randomNumber);
        }
        ]]>
    </fx:Script>
    <mx:HSlider id="slider1" tickInterval="1"
        labels="[1,2,3,4,5,6,7,8,9,10]" width="282"/>
    <s:Button id="myButton"
        label="Change Thumb Position"
        click="adjustThumb(slider1);"/>
</s:Application>
```

To invoke a method from a child document (such as a custom MXML component), you can use the `parentApplication`, `parentDocument`, or `FlexGlobals.topLevelApplication` properties. For more information, see "Application containers" on page 393.

***Note:** Because Flex invokes the `initialize` event before drawing the component, you cannot access size and position information of that component from within the `initialize` event handler unless you use the `creationComplete` event handler. For more information on the order of initialization events, see "About startup order" on page 2334.*

Creating visual components in ActionScript

You can use ActionScript to programmatically create visual components using the `new` operator, in the same way that you create instances of any ActionScript class. The created component has default values for its properties, but it does not yet have a parent or any children (including any kind of internal DisplayObjects), and it is not yet on the display list in Flash Player or Adobe® AIR™, so you can't see it. After creating the component, you should use standard assignment statements to set any properties whose default values aren't appropriate.

Finally, you must add the new component to a container, by using the Spark container's `addElement()` or `addElementAt()` methods, so that it becomes part of the visual hierarchy of an application. (For MX containers, you can use the `addChild()` or `addChildAt()` methods.) The first time that it is added to a container, a component's children are created. Children are created late in the component's life cycle so that you can set properties that can affect children as they are created.

When creating visual controls, you must import the appropriate package. In most cases, this is the mx.controls package, although you should check the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The following example creates a Button control inside the HGroup container:

```
<?xml version="1.0"?>
<!-- usingas/ASVisualComponent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import spark.components.Button;
        public var button2:Button;
        public function createObject():void {
            button2 = new Button();
            button2.label = "Click Me";
            hb1.addElement(button2);
        }
        ]]>
    </fx:Script>
    <s:HGroup id="hb1">
        <s:Button label="Create Object" click="createObject()"/>
    </s:HGroup>
</s:Application>
```

Flex creates the new child as the last child in the container. If you do not want the new child to be the last in the Spark container, use the `addElementAt()` method to change the order. You can use the `setItemIndex()` method after the call to the `addElement()` method, but this is less efficient. For MX containers, you use the `addChildAt()`, `setChildIndex()`, and `addChild()` methods.

You should declare an instance variable for each dynamically created component and store a reference to the newly created component in it, just as the MXML compiler does when you set an `id` property for a component instance tag. You can then access your dynamically created components in the same way as those declaratively created in MXML.

To programmatically remove a control in Spark containers, you can use the `removeElement()`, `removeElementAt()`, and `removeAllElements()` methods. For MX containers, you use the `removeChild()` or `removeChildAt()` methods. You can also use the `removeAllChildren()` method to remove all child controls from a container.

Calling the "remove" methods does not actually delete the objects from memory. If you do not have any other references to the child, Flash Player includes the object in garbage collection at some future point. But if you have a reference to that child, the child is not garbage collected.

In some cases, you declaratively define a component with an MXML tag. You can set the `creationPolicy` property of the component's container to `none` to defer the creation of the controls inside that container. You can then create the component programmatically rather than declaratively. For information on using the `creationPolicy` property, see "Improving startup performance" on page 2333.

The only component you can pass to the `addElement()` method is a class that implements the IVisualElement interface. In other words, if you create a new object that is not a subclass of mx.core.IVisualElement, you must wrap it in a class that implments IVisualElement before you can attach it to a container. The following example creates a new Sprite object, which is not a subclass of IVisualElement, and adds it as a child of the UIComponent (which implements IVisualElement) before adding it to the Panel container:

```
<?xml version="1.0"?>
<!-- usingas/AddingChildrenAsUIComponents.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import flash.display.Sprite;
        import mx.core.UIComponent;
        private var xLoc:int = 20;
        private var yLoc:int = 20;
        private var circleColor:Number = 0xFFCC00;
        private function addChildToPanel():void {
            var circle:Sprite = new Sprite();
            circle.graphics.beginFill(circleColor);
            circle.graphics.drawCircle(xLoc, yLoc, 15);
            var c:UIComponent = new UIComponent();
            c.addChild(circle);
            panel1.addElement(c);

            xLoc = xLoc + 5;
            yLoc = yLoc + 1;
            circleColor = circleColor + 20;
        }
        ]]>
    </fx:Script>
    <s:Panel id="panel1" height="250" width="300"/>
    <s:Button id="myButton" label="Click Me" click="addChildToPanel();"/>

</s:Application>
```

About scope

Scoping in ActionScript is largely a description of what the `this` keyword refers to at a particular point. In your application’s core MXML file, you can access the Application object by using the `this` keyword. In a file defining an MXML component, `this` is a reference to the current instance of that component.

In an ActionScript class file, the `this` keyword refers to the instance of that class. In the following example, the `this` keyword refers to an instance of myClass. Because `this` is implicit, you do not have to include it, but it is shown here to illustrate its meaning.

```
class myClass {
    var _x:Number = 3;
    function get x():Number {
        return this._x;
    }
    function set x(y:Number):void {
        if (y > 0) {
            this._x = y;
        } else {
            this._x = 0;
        }
    }
}
```

However, in custom ActionScript and MXML components or external ActionScript class files, Flex executes in the context of those objects and classes, and the `this` keyword refers to the current scope and not the Application object scope.

Flex includes a `FlexGlobals.topLevelApplication` property that you can use to access the root application. In some cases, you can also use the `parentDocument` property to access the next level up in the document chain of an application, or the `parentApplication` property to access the next level up in the application chain when one Application object loads another Application object.

You cannot use these properties to access the root application if the loaded application was loaded into a separate ApplicationDomain or SecurityDomain, as is the case with sandboxed and multi-versioned applications. For more information, see “Accessing the main application from sub-applications” on page 197.

If you write ActionScript in a component’s event listener, the scope is not the component but rather the application. For example, the following code changes the label of the Button control to `"Clicked"` once the Button control is pressed:

```
<?xml version="1.0"?>
<!-- usingas/ButtonScope.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:BasicLayout/>
    </s:layout>
    <s:Panel width="250" height="100" x="65" y="24">
        <s:Button id="myButton"
            label="Click Me"
            click="myButton.label='Clicked'"
            x="79.5" y="20"/>
    </s:Panel>
    <s:Button label="Reset"
        x="158" y="149"
        click="myButton.label='Click Me'"/>
</s:Application>
```

Contrast the previous example with the following code:

```
<?xml version="1.0"?>
<!-- usingas/AppScope.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <!-- The following throws a compiler error because the app level scope does
         not have a label to set. -->
    <!-- <s:Button id="myButton" label="Click Me" click="{this.label='Clicked'}"/> -->
    <!-- <s:Button label="Reset" click="myButton.label='Click Me'"/> -->
</s:Application>
```

This code does not work because when an event listener executes, the `this` keyword does not refer to the Button instance; the `this` keyword refers to the Application or other top-level component instance. The second example attempts to set the `label` property of the Application object, not the `label` property of the Button.

Variables declared within a function are locally scoped to that function. These variables can share the same name as variables in outer scopes, and they do not affect the outer-scoped variable. If a variable is just used temporarily by a single method, make it a local variable of that method rather than an instance variable. Use instance variables only for storing the state of an instance, because each instance variable will take up memory for the entire lifetime of the instance. You can refer to the outer-scoped variable with the `this.` prefix.

Including versus importing ActionScript code

To make your MXML code more readable, you can reference ActionScript files in your `<fx:Script>` tags, rather than insert large blocks of script. You can either include or import ActionScript files.

There is a distinct difference between including and importing code in ActionScript. *Including* copies lines of code from one file into another, as if they had been pasted at the position of the `include` statement. *Importing* adds a reference to a class file or package so that you can access objects and properties defined by external classes. Files that you import must be found in the source path. Files that you include must be located relative to the file that uses the `include` statement, or you must use an absolute path.

You use the `include` statement or the `<fx:Script source="`*`filename`*`">` tag to add ActionScript code to your applications.

You use `import` statements in an `<fx:Script>` block to define the locations of ActionScript classes and packages that your applications might use.

Including ActionScript files

To include ActionScript code, you reference an external ActionScript file in your `<fx:Script>` tags. At compile time, the compiler copies the entire contents of the file into your MXML application, as if you had actually typed it. As with ActionScript in an `<fx:Script>` block, ActionScript statements can only be inside functions. Included files can also declare constants and namespaces, include other ActionScript files, import declarations, and use namespaces. You cannot define classes in included files.

Variables and functions defined in an included ActionScript file are available to any component in the MXML file. An included ActionScript file is not the same as an imported ActionScript class. Flex provides access to the included file's variables and functions, but does not add a new class, because the MXML file itself is a class.

Included ActionScript files do not need to be in the same directory as the MXML file. However, you should organize your ActionScript files in a logical directory structure.

There are two ways to include an external ActionScript file in your application:

- The `source` attribute of the `<fx:Script>` tag. This is the preferred method for including external ActionScript class files.
- The `include` statement inside `<fx:Script>` blocks.

Using the source attribute to include ActionScript files

You use the `source` attribute of the `<fx:Script>` tag to include external ActionScript files in your applications. This provides a way to make your MXML files less cluttered and promotes code reuse across different applications.

Do not give the script file the same name as the application file. This causes a compiler error.

The following example shows the contents of the IncludedFile.as file:

```
// usingas/includes/IncludedFile.as
public function computeSum(a:Number, b:Number):Number {
    return a + b;
}
```

The following example uses the `<fx:Script>` tag to include the contents of the IncludedFile.as file. This file is located in the /includes subdirectory.

```
<?xml version="1.0"?>
<!-- usingas/SourceInclude.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script source="includes/IncludedFile.as"/>
    <s:TextInput id="ta1st" text="3" width="40" x="170" y="24" textAlign="right"/>
    <s:TextInput id="ta2nd" text="3" width="40" x="170" y="52" textAlign="right"/>
    <s:TextArea id="taMain" height="25" width="78" x="132" y="82" textAlign="right"/>
    <s:Button id="b1" label="Compute Sum"
        click="taMain.text=String(computeSum(Number(ta1st.text), Number(ta2nd.text)));"
        x="105" y="115"/>
    <s:Label x="148" y="52" text="+" fontWeight="bold" fontSize="17" width="23"/>
</s:Application>
```

The `source` attribute of the `<fx:Script>` tag supports both relative and absolute paths. For more information, see "Referring to external files that have been included" on page 46.

You cannot use the `source` attribute of an `<fx:Script>` tag and wrap ActionScript code inside that same `<fx:Script>` tag. To include a file and write ActionScript in the MXML file, use two `<fx:Script>` tags.

Using the include directive

The `include` directive is an ActionScript statement that copies the contents of the specified file into your MXML file. The `include` directive uses the following syntax:

```
include "file_name";
```

The following example includes the myfunctions.as file:

```
<?xml version="1.0"?>
<!-- usingas/IncludeASFile.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        /*  The myfunctions.as file defines two methods that
            return Strings. */
        include "includes/myfunctions.as";
        ]]>
    </fx:Script>
    <s:Button id="myButton"
        label="Call Methods in Included File"
        click="ta1.text=doSomething();ta1.text+=doSomethingElse()"/>
    <s:TextArea width="268" id="ta1"/>
    <s:Button label="Clear" click="ta1.text=''"/>
</s:Application>
```

You can specify only a single file for each `include` directive, but you can use any number of `include` directives. You can nest `include` directives; files with `include` directives can include files that have `include` directives.

The `include` directive supports only relative paths. For more information, see "Referring to external files that have been included" on page 46.

You can use the `include` only where multiple statements are allowed. For example, the following is not allowed:

```
if (expr)
    include "foo.as"; // First statement is guarded by IF, but rest are not.
...
```

The following is allowed:

```
if (expr) {
    include "foo.as"; // All statements inside { } are guarded by IF.
}
```

The use of curly braces ({ }) allows multiple statements because you can add multiple statements inside the braces.

Adobe recommends that you not use the `include` directive if you use a large number of included ActionScript files. You should try to break the code into separate class files where appropriate and store them in logical package structures.

Referring to external files that have been included

The `source` attribute of the `<fx:Script>` tag and the `include` directive refer to files in different ways.

The following are the valid paths to external files that are referenced in an `<fx:Script>` tag's `source` attribute:

- Absolute URLs, such as http://www.macromedia.com or file:///C|/site_flashteam/foo.gif.
- Relative URLs, such as ../myscript.as. A relative URL that does not start with a slash is resolved relative to the file that uses it. If the tag `<fx:Script source="../IncludedFile.as">` is included in "mysite/myfiles/myapp.mxml," the system searches for "mysite/IncludedFile.as".

For an ActionScript `include` directive, you can reference only relative URLs.

Flex searches the source path for imported classes and packages. Flex does not search the source path for files that are included using the `include` directive or the `source` attribute of the `<fx:Script>` tag.

Importing classes and packages

If you create many utility classes or include multiple ActionScript files to access commonly used functions, you might want to store them in a set of classes in their own package. You can import ActionScript classes and packages using the `import` statement. By doing this, you do not have to explicitly enter the fully qualified class names when accessing classes within ActionScript.

The following example imports the MyClass class in the MyPackage.Util package:

```
<?xml version="1.0"?>
<!-- usingas/AccessingPackagedClasses.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import MyPackage.Util.MyClass;
            private var mc:MyClass = new MyClass;
        ]]>
    </fx:Script>
    <s:Button id="myButton" label="Click Me" click="myButton.label=mc.returnAString()"/>
</s:Application>
```

In your ActionScript code, instead of referring to the class with its fully qualified package name (MyPackage.Util.MyClass), you refer to it as MyClass.

You can also use the wildcard character (*) to import the entire package. For example, the following statement imports the entire MyPackage.Util package:

```
import MyPackage.Util.*;
```

Flex searches the source path for imported files and packages, and includes only those that are used in the final SWF file.

It is not sufficient to simply specify the fully qualified class name. You should use fully qualified class names only when necessary to distinguish two classes with the same class name that reside in different packages.

If you import a class but do not use it in your application, the class is not included in the resulting SWF file's bytecode. As a result, importing an entire package with a wildcard does not create an unnecessarily large SWF file.

Techniques for separating ActionScript from MXML

The following sample application, which calls a single function, shows several methods of separating ActionScript from the MXML.

The Temperature application takes input from a single input field and uses a function to convert the input from Fahrenheit to Celsius. It then displays the resulting temperature in a Label control.

One MXML document (event handling logic in event attribute)

The following code shows the ActionScript event handling logic inside the MXML tag's `click` event:

```
<?xml version="1.0"?>
<!-- usingas/ASOneFile.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="700">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Temperature Application" width="90%">
        <s:HGroup>
            <s:Label text="Temperature in Fahrenheit:"/>
            <s:TextInput id="fahrenheit" width="120"/>
            <s:Button label="Convert"
                click="celsius.text=String(Math.round((Number(fahrenheit.text)-32)/1.8 *
10)/10);"/>
            <s:Label text="Temperature in Celsius:"/>
            <s:Label id="celsius" width="120" fontSize="24"/>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

One MXML document (event handling logic in <fx:Script> block)

In this example, the logic for the function is inside an `<fx:Script>` block in the MXML document, and is called from the MXML tag's `click` event, as the following code shows:

```
<?xml version="1.0"?>
<!-- usingas/ASScriptBlock.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="700">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        public function calculate():void {
            var n:Number = Number(fahrenheit.text);
            var t:Number = Math.round((n-32)/1.8*10)/10;
            celsius.text=String(t);
        }
        ]]>
    </fx:Script>
    <s:Panel title="Temperature Application" width="90%">
        <s:HGroup>
            <s:Label text="Temperature in Fahrenheit:"/>
            <s:TextInput id="fahrenheit" width="120"/>
            <s:Button label="Convert" click="calculate();" />
            <s:Label text="Temperature in Celsius:"/>
            <s:Label id="celsius" width="120" fontSize="24"/>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

One MXML document and one ActionScript file (event handling logic in separate script file)

Here, the function call is in an MXML event attribute, and the function is defined in a separate ActionScript file, as the following code shows:

```
<?xml version="1.0"?>
<!-- usingas/ASSourceFile.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="700">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <!-- Specify the ActionScript file that contains the function. -->
    <fx:Script source="includes/Sample3Script.as"/>
    <s:Panel title="Temperature Application" width="90%">
        <s:HGroup>
            <s:Label text="Temperature in Fahrenheit:"/>
            <s:TextInput id="fahrenheit" width="120"/>
            <s:Button label="Convert" click="celsius.text=calculate(fahrenheit.text);"/>
            <s:Label text="Temperature in Celsius:"/>
            <s:Label id="celsius" width="120" fontSize="24"/>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

The Sample3Script.as ActionScript file contains the following code:

```
// usingas/includes/Sample3Script.as
public function calculate(s:String):String {
    var n:Number = Number(s);
    var t:Number = Math.round((n-32)/1.8*10)/10;
    return String(t);
}
```

Creating ActionScript components

You can create reusable components that use ActionScript and reference these components in your applications as MXML tags. Components created in ActionScript can contain graphical elements, define custom business logic, or extend existing components. They can inherit from any components available in Flex.

Defining your own components in ActionScript has several benefits. Components let you divide your applications into individual modules that you can develop and maintain separately. By implementing commonly used logic within custom components, you can build a suite of reusable components that you can share among multiple applications.

Also, you can base your custom components on the set of components by extending from the Flex class hierarchy. You can create custom versions of Flex visual controls, as well as custom versions on nonvisual components, such as data validators, formatters, and effects.

For example, you can define a custom button, derived from the Button control, in the myControls package, as the following example shows:

```
package myControls {
    import mx.controls.Button;
    public class MyButton extends Button {
        public function MyButton() {
            ...
        }
        ...
    }
}
```

In this example, you write your MyButton control to the MyButton.as file, and you store the file in the myControls subdirectory of the root directory of your application. The fully qualified class name of your component reflects its location. In this example, the component's fully qualified class name is myControls.MyButton.

You can reference your custom Button control from an application file, such as MyApp.mxml, as the following example shows:

```
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:cmp="myControls.*">
    <cmp:MyButton label="Jack"/>
</s:Application>
```

In this example, you define the `cmp` namespace that defines the location of your custom component in the application's directory structure. You then reference the component as an MXML tag using the namespace prefix.

Typically, you put custom ActionScript components in directories that are in the source path. These include any directory that you specify in the source path.

You can also create custom components using MXML. For more information, see "Custom components" on page 2356.

Types of custom components

You can create the following types of components in ActionScript:

User-interface components *User-interface components* contain both processing logic and visual elements. These components usually extend the component hierarchy. You can extend from the UIComponent classes, or any of the components, such as Button, ComboBox, or DataGrid. Your custom ActionScript component inherits all of the public methods, along with public and protected properties of its base class.

Nonvisual components *Nonvisual components* define no visual elements. A nonvisual component is an ActionScript class that does not extend the UIComponent class. They can provide greater efficiency at run time.

Performing object introspection

Object introspection is a technique for determining the elements of a class at run time, such as its properties and methods. There are two ways to do introspection in ActionScript:

- Using `for..in` loops
- Using the introspection API

You might find object introspection a useful technique when debugging your application. For example, you might write a method that takes a generic object of type Object as an argument. You can use introspection to output all of the properties and methods of the Object to determine exactly what your application passed to it.

Using for..in loops

You can use a `for..in` loop to iterate over objects and output their properties and their values. A `for..in` loop enumerates only dynamically added properties. Declared variables and methods of classes are not enumerated in `for..in` loops. This means that most classes in the ActionScript API will not display any properties in a `for..in` loop. However, the generic type Object is still a dynamic object and will display properties in a `for..in` loop.

The following example creates a generic Object, adds properties to that object, and then iterates over that object when you click the button to inspect its properties:

```
<?xml version="1.0"?>
<!-- usingas/IntrospectionForIn.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        private var obj:Object = new Object();
        private function initApp():void {
            // Create the object.
            obj.a = "Schotten Totten";
            obj.b = "Taj Majal";
            obj.c = "Durche die Wuste";
        }
        public function dumpObj():void {
            for (var p:String in obj) {
                ta1.text += p + ":" + obj[p] + "\n";
            }
        }
        ]]>
    </fx:Script>
    <s:TextArea id="ta1" width="400" height="200"/>
    <s:Button label="Dump Object" click="dumpObj()"/>
</s:Application>
```

You can also use the `mx.utils.ObjectUtil.toString()` method to print all the dynamically added properties of an object, for example:

```
<?xml version="1.0"?>
<!-- usingas/ObjectUtilToString.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.utils.ObjectUtil;
        private var obj:Object = new Object();
        private function initApp():void {
            // Create the object.
            obj.a = "Schotten Totten";
            obj.b = "Taj Majal";
            obj.c = "Durche die Wuste";
        }
        public function dumpObj():void {
            ta1.text = ObjectUtil.toString(obj);
        }
        ]]>
    </fx:Script>
    <s:TextArea id="ta1" width="400" height="200"/>
    <s:Button label="Dump Object" click="dumpObj()"/>
</s:Application>
```

The mx.utils.ObjectUtil class has other useful methods such as `compare()`, `copy()`, and `isSimple()`. For more information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Using the introspection API

If you want to list all the public properties and methods of a nondynamic (or sealed) class or class instance, use the `describeType()` method and parse the results using the E4X API. The `describeType()` method is in the flash.utils package. The method's only parameter is the target object that you want to introspect. You can pass it any ActionScript value, including all available ActionScript types such as object instances, primitive types such as `uint`, and class objects. The return value of the `describeType()` method is an E4X XML object that contains an XML description of the object's type.

The `describeType()` method returns only public members. The method does not return private members of the caller's superclass or any other class where the caller is not an instance. If you call `describeType(this)`, the method returns information only about nonstatic members of the class. If you call `describeType(getDefinitionByName("MyClass"))`, the method returns information only about the target's static members.

The following example introspects the Button control and prints the details to TextArea controls:

```
<?xml version="1.0"?>
<!-- usingas/IntrospectionAPI.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="getDetails()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
        import flash.utils.*;
        public function getDetails():void {
            // Get the Button control's E4X XML object description.
            var classInfo:XML = describeType(button1);
            // Dump the entire E4X XML object into ta2.
            ta2.text = classInfo.toString();
            // List the class name.
            ta1.text = "Class " + classInfo.@name.toString() + "\n";
            // List the object's variables, their values, and their types.
            for each (var v:XML in classInfo..variable) {
                ta1.text += "Variable " + v.@name + "=" + button1[v.@name] +
                    " (" + v.@type + ")\n";
            }
            // List accessors as properties.
            for each (var a:XML in classInfo..accessor) {
                // Do not get the property value if it is write only.
                if (a.@access == 'writeonly') {
                    ta1.text += "Property " + a.@name + " (" + a.@type +")\n";
                }
                else {
                    ta1.text += "Property " + a.@name + "=" +
                        button1[a.@name] +  " (" + a.@type +")\n";
                }
            }
            // List the object's methods.
            for each (var m:XML in classInfo..method) {
                ta1.text += "Method " + m.@name + "():" + m.@returnType + "\n";
            }
        }
        ]]>
    </fx:Script>
    <s:Button label="This Button Does Nothing" id="button1"/>
    <s:TextArea id="ta1" width="400" height="200"/>
    <s:TextArea id="ta2" width="400" height="200"/>
</s:Application>
```

The output displays accessors, variables, and methods of the Button control, and appears similar to the following:

```
Class mx.controls::Button
...
Variable id=button1 (String)
Variable __width=66 (Number)
Variable layoutWidth=66 (Number)
Variable __height=22 (Number)
Variable layoutHeight=22 (Number)
...
Property label=Submit (String)
Property enabled=true (Boolean)
Property numChildren=2 (uint)
Property enabled=true (Boolean)
Property visible=true (Boolean)
Property toolTip=null (String)
...
Method dispatchEvent():Boolean
Method hasEventListener():Boolean
Method layoutContents():void
Method getInheritingStyle():Object
Method getNonInheritingStyle():Object
```

Another useful method is the ObjectUtil's `getClassInfo()` method. This method returns an Object with the name and properties of the target object. The following example uses the `getClassInfo()` and `toString()` methods to show the properties of the Button control:

```
<?xml version="1.0"?>
<!-- usingas/IntrospectionObjectUtil.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="650">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import mx.controls.Alert;
        import mx.utils.ObjectUtil;
        private function showProps(b:Button):void {
            var o:Object = ObjectUtil.getClassInfo(b);
            ta1.text = ObjectUtil.toString(o);
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Show Properties" click="showProps(b1)"/>
    <s:TextArea id="ta1" width="300" height="500"/>
</s:Application>
```

For more information about using E4X, see Working with XML.

Events

One of the most important parts of your Adobe® Flex™ application is handling events by using controls and ActionScript.

About Flex events

Events let a developer know when something happens within an application. They can be generated by user devices, such as the mouse and keyboard, or other external input, such as the return of a web service call. Events are also triggered when changes happen in the appearance or life cycle of a component, such as the creation or destruction of a component or when the component is resized.

Any user interaction with your application can generate events. Events can also occur without any direct user interaction, such as when data finishes loading from a server or when an attached camera becomes active. You can “handle” these events in your code by adding an event handler. *Event handlers* are the functions or methods that you write to respond to specific events. They are also sometimes referred to as *event listeners*.

The Flex event model is based on the Document Object Model (DOM) Level 3 events model. Although Flex does not adhere specifically to the DOM standard, the implementations are very similar. The event model in Flex comprises the Event object and its subclasses, and the event dispatching model. For a quick start in using events in Flex, see the sample code in “Using events” on page 57.

Components generate and dispatch events and *consume* (listen to) other events. An object that requires information about another object’s events registers a listener with that object. When an event occurs, the object dispatches the event to all registered listeners by calling a function that was requested during registration. To receive multiple events from the same object, you must register your listener for each event.

Components have built-in events that you can handle in ActionScript blocks in your MXML applications. You can also take advantage of the Flex event system’s dispatcher-listener model to define your own event listeners outside of your applications, and define which methods of your custom listeners will listen to certain events. You can register listeners with the target object so that when the target object dispatches an event, the listeners get called.

All visual objects, including Flex controls and containers, are subclasses of the DisplayObject class. They are in a tree of visible objects that make up your application. The root of the tree is the Stage. Below that is the SystemManager object, and then the Application object. Child containers and components are leaf nodes of the tree. That tree is known as the *display list*. An object on the display list is analogous to a node in the DOM hierarchical structure. The terms *display list object* and *node* are used interchangeably.

For information about each component’s events, see the component’s description in “UI Controls” on page 643 or the control’s entry in *ActionScript 3.0 Reference for the Adobe Flash Platform*.

For a detailed description of a component’s startup life cycle, including major events in that life cycle, see “Create advanced MX visual components in ActionScript” on page 2475.

About the event flow

You can instruct any container or control to listen for events dispatched by another container or control. When Adobe® Flash® Player dispatches an Event object, that Event object makes a roundtrip journey from the root of the display list to the target node, checking each node for registered listeners. The *target node* is the node in the display list where the event occurred. For example, if a user clicks a Button control named Child1, Flash Player dispatches an Event object with Child1 defined as the target node.

The event flow is conceptually divided into three parts: the capturing phase, the targeting phase, and the bubbling phase, as briefly described next. For more information about the event flow, see “Event propagation” on page 81.

About the capturing phase

The first part of the event flow is called the *capturing phase*. This phase comprises all of the nodes from the root node to the parent of the target node. During this phase, Flash Player examines each node, starting with the root, to see if it has a listener registered to handle the event. If it does, Flash Player sets the appropriate values of the Event object and then calls that listener. Flash Player stops after it reaches the target node's parent and calls any listeners registered on the parent. For more information, see "Capturing phase" on page 82.

About the targeting phase

The second part of the event flow, the *targeting phase*, consists solely of the target node. Flash Player sets the appropriate values on the Event object, checks the target node for registered event listeners, and then calls those listeners. For more information, see "Targeting phase" on page 83.

About the bubbling phase

The third part of the event flow, the *bubbling phase*, comprises all of the nodes from the target node's parent to the root node. Starting with the target node's parent, Flash Player sets the appropriate values on the Event object and then calls event listeners on each of these nodes. Flash Player stops after calling any listeners on the root node. For more information about the bubbling phase, see "Bubbling phase" on page 83.

About the Event class

The Event class is an ActionScript class with properties that contain information about the event that occurred. An Event object is an implicitly created object, similar to the request and response objects in a JavaServer Page (JSP) that are implicitly created by the application server.

Flex creates an Event object each time an event is dispatched. You can use the Event object inside an event listener to access details about the event that was dispatched, or about the component that dispatched the event. Passing an Event object to, and using it in, an event listener is optional. However, if you want to access the Event object's properties inside your event listeners, you must pass the Event object to the listener.

Flex creates only one Event object when an event is dispatched. During the bubbling and capturing phases, Flex changes the values on the Event object as it moves up or down the display list, rather than creating a new Event object for each node.

About event subclasses

There are many classes that extend the flash.events.Event class. These classes are defined mostly in the following packages:

- spark.events.*
- mx.events.*
- flash.events.*

The mx.events package defines event classes that are specific to most Flex controls, including the DataGridEvent, DragEvent, and ColorPickerEvent. The spark.events package defines event classes that are specific to a few Spark controls, including the TextOperationEvent and VideoEvent. The flash.events package describes events that are not unique to Flex but are instead defined by Flash Player. These event classes include MouseEvent, DataEvent, and TextEvent. All of these events are commonly used in applications.

In addition to these packages, some packages also define their own event objects: for example, mx.messaging.events.ChannelEvent and mx.logging.LogEvent.

Child classes of the Event class have additional properties and methods that may be unique to them. In some cases, you will want to use a more specific event type rather than the generic Event object so that you can access these unique properties or methods. For example, the LogEvent class has a `getLevelString()` method that the Event class does not.

For information on using Event subclasses, see “Using event subclasses” on page 89.

About the EventDispatcher class

Every object in the display list can trace its class inheritance back to the DisplayObject class. The DisplayObject class, in turn, inherits from the EventDispatcher class. The EventDispatcher class is a base class that provides important event model functionality for every object on the display list. Because the DisplayObject class inherits from the EventDispatcher class, any object on the display list has access to the methods of the EventDispatcher class.

This is significant because every item on the display list can participate fully in the event model. Every object on the display list can use its `addEventListener()` method—inherited from the EventDispatcher class—to listen for a particular event, but only if the listening object is part of the event flow for that event.

Although the name EventDispatcher seems to imply that this class’s main purpose is to send (or dispatch) Event objects, the methods of this class are used much more frequently to register event listeners, check for event listeners, and remove event listeners.

The EventDispatcher class implements the IEventDispatcher interface. This allows developers who create custom classes that cannot inherit from EventDispatcher or one of its subclasses to implement the IEventDispatcher interface to gain access to its methods.

The `addEventListener()` method is the most commonly used method of this class. You use it to register your event listeners. For information on using the `addEventListener()` method, see “Using the addEventListener() method” on page 64.

Advanced programmers use the `dispatchEvent()` method to manually dispatch an event or to send a custom Event object into the event flow. For more information, see “Manually dispatching events” on page 78.

Several other methods of the EventDispatcher class provide useful information about the existence of event listeners. The `hasEventListener()` method returns `true` if an event listener is found for that specific event type on a particular display list object. The `willTrigger()` method checks for event listeners on a particular display list object, but it also checks for listeners on all of that display list object’s ancestors for all phases of the event flow. The method returns `true` if it finds one.

Using events

Using events in Flex is a two-step process. First, you write a function or class method, known as an *event listener* or *event handler,* that responds to events. The function often accesses the properties of the Event object or some other settings of the application state. The signature of this function usually includes an argument that specifies the event type being passed in.

The following example shows a simple event listener function that reports when a control triggers the event that it is listening for:

```
<?xml version="1.0"?>
<!-- events/SimpleEventHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function initApp():void {
            b1.addEventListener(MouseEvent.CLICK, myEventHandler);
        }
        private function myEventHandler(event:Event):void {
            Alert.show("An event occurred.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"/>
</s:Application>
```

As you can see in this example, you also register that function or class method with a display list object by using the addEventListener() method.

Most Flex controls simplify listener registration by letting you specify the listener inside the MXML tag. For example, instead of using the `addEventListener()` method to specify a listener function for the Button control's `click` event, you specify it in the `click` attribute of the `<mx:Button>` tag:

```
<?xml version="1.0"?>
<!-- events/SimplerEventHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function myEventHandler(event:Event):void {
            Alert.show("An event occurred.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</s:Application>
```

This is equivalent to the `addEventListener()` method in the previous code example. However, it is best practice to use the `addEventListener()` method. This method gives you greater control over the event by letting you configure the priority and capturing settings, and use event constants. In addition, if you use `addEventListener()` to add an event handler, you can use `removeEventListener()` to remove the handler when you no longer need it. If you add an event handler inline, you cannot call `removeEventListener()` on that handler.

Each time a control generates an event, Flex creates an Event object that contains information about that event, including the type of event and a reference to the dispatching control. To use the Event object, you specify it as a parameter in the event handler function, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/EventTypeHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function myEventHandler(e:Event):void {
            Alert.show("An event of type '" + e.type + "' occurred.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</s:Application>
```

If you want to access the Event object in an event handler that was triggered by an inline event, you must add the `event` keyword inside the MXML tag so that Flex explicitly passes it to the handler, as in the following:

```
<mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
```

You are not required to use the Event object in a handler function. The following example creates two event handler functions and registers them with the events of a ComboBox control. The first event handler, `openEvt()`, takes no arguments. The second event handler, `changeEvt()`, takes the Event object as an argument and uses this object to access the `value` and `selectedIndex` of the ComboBox control that triggered the event.

```
<?xml version="1.0"?>
<!-- events/MultipleEventHandlers.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function openEvt():void {
            forChange.text="";
        }
        private function changeEvt(e:Event):void {
            forChange.text =
            "Value: " + e.currentTarget.selectedItem + "\n" +
            "Index: " + e.currentTarget.selectedIndex;
        }
    ]]></fx:Script>
    <s:ComboBox open="openEvt()" change="changeEvt(event)">
        <s:dataProvider>
            <s:ArrayList>
                <fx:String>AK</fx:String>
                <fx:String>AL</fx:String>
                <fx:String>AR</fx:String>
            </s:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
    <s:TextArea id="forChange" width="150" height="100"/>
</s:Application>
```

This example shows accessing the `target` property of the Event object. For more information, see “Accessing the currentTarget property” on page 60.

Specifying the Event object

You specify the object in a listener function’s signature as type Event, as the following example shows:

```
function myEventListener(e:Event):void { ... }
```

However, if you want to access properties that are specific to the type of event that was dispatched, you must instead specify a more specific event type, such as ToolTipEvent or KeyboardEvent, as the following example shows:

```
import mx.events.ToolTip
function myEventListener(e:ToolTipEvent):void { ... }
```

In some cases, you must import the event’s class in your ActionScript block.

Most objects have specific events that are associated with them, and most of them can dispatch more than one type of event.

If you declare an event of type Event, you can cast it to a more specific type to access its event-specific properties. For more information, see “Using event subclasses” on page 89.

Accessing the currentTarget property

Event objects include a reference to the instance of the dispatching component (or *target*), which means that you can access all the properties and methods of that instance in an event listener. The following example accesses the `id` of the Button control that triggered the event:

```
<?xml version="1.0"?>
<!-- events/AccessingCurrentTarget.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function myEventHandler(e:Event):void {
                Alert.show("The button '" + e.currentTarget.id + "' was clicked.");
            }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</s:Application>
```

You can access members of the `currentTarget`. If you do not cast the current target to a specific type, the compiler assumes that it is of type Object. Objects can have any property or method because the Object type is dynamic in ActionScript. Therefore, when accessing methods and properties of the `currentTarget`, it is best practice to cast `currentTarget` to whatever class you anticipate will dispatch that event. This gives you strong type checking at compile time, and helps avoid the risk of throwing a run-time error.

The following example casts the current target to a TextInput class before calling the `selectRange()` method, but does not cast it before trying to set the `tmesis` property. The `tmesis` property does not exist on the TextInput class. This illustrates that you will get a run-time error but not a compile-time error when you try to access members that don’t exist, unless you cast `currentTarget` to a specific type so that type checking can occur:

```
<?xml version="1.0"?>
<!-- events/InvokingOnCurrentTarget.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="500">
    <fx:Script>
        <![CDATA[
        import mx.core.UIComponent;
        private function tiHandler(e:Event):void {
            /*
            The following enforces type checking:
            */
            TextInput(e.currentTarget).selectRange(0,3);
            /*
            The following throws a run-time error but not a compile-time error:
            e.currentTarget.tmesis = 4;
            */
            /*
            ... unless you cast it to the expected type like the following. Then
            the compiler throws an error.
            TextInput(e.currentTarget).tmesis = 4;
            */
        }
        ]]>
    </fx:Script>
    <s:TextInput id="ti1" click="tiHandler(event)"
        text="When you click on this control, the first three characters are selected."
        width="400"/>
</s:Application>
```

You could also cast `currentTarget` to UIComponent or some other more general class that still has methods of display objects. That way, if you don't know exactly which control will dispatch an event, at least you can ensure there is some type checking.

You can also access methods and properties of the `target` property, which contains a reference to the current node in the display list. For more information, see "About the target and currentTarget properties" on page 81.

Registering event handlers

There are several strategies that you can employ when you register event handlers with your Flex controls:

1 Define an event handler inline. This binds a call to the handler function to the control that triggers the event.

```
<?xml version="1.0"?>
<!-- events/SimplerEventHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function myEventHandler(event:Event):void {
            Alert.show("An event occurred.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</s:Application>
```

In this example, whenever the user clicks the Button control, Flex calls the `myClickHandler()` function.

For more information on defining event handlers inline, see “Defining event listeners inline” on page 63.

2 Use the `addEventListener()` method, as follows:

```
<?xml version="1.0"?>
<!-- events/SimpleEventHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function initApp():void {
            b1.addEventListener(MouseEvent.CLICK, myEventHandler);
        }
        private function myEventHandler(event:Event):void {
            Alert.show("An event occurred.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"/>
</s:Application>
```

As with the previous example, whenever the user clicks the Button control, Flex calls the `myClickHandler()` handler function. However, registering your event handlers using this method provides more flexibility. You can register multiple components with this event handler, add multiple handlers to a single component, or remove the handler. For more information, see “Using the addEventListener() method” on page 64.

3 Create an event handler class and register components to use the class for event handling. This approach to event handling promotes code reuse and lets you centralize event handling outside your MXML files. For more information on creating custom event handler classes, see “Creating event handler classes” on page 69.

Defining event listeners inline

The simplest method of defining event handlers in applications is to point to a handler function in the component's MXML tag. To do this, you add any of the component's events as a tag attribute followed by an ActionScript statement or function call.

You add an event handler inline using the following syntax:

```
<s:tag_nameevent_name="handler_function"/>
```

For example, to listen for a Button control's `click` event, you add a statement in the `<mx:Button>` tag's `click` attribute. If you add a function, you define that function in an ActionScript block. The following example defines the `submitForm()` function as the handler for the Button control's `click` event:

```
<fx:Script><![CDATA[
    function submitForm():void {
        // Do something.
    }
]]></fx:Script>
<s:Button label="Submit" click="submitForm();"/>
```

Event handlers can include any valid ActionScript code, including code that calls global functions or sets a component property to the return value. The following example calls the `trace()` global function:

```
<s:Button label="Get Ver" click="trace('The button was clicked');"/>
```

There is one special parameter that you can pass in an inline event handler definition: the `event` parameter. If you add the `event` keyword as a parameter, Flex passes the Event object and inside the handler function, you can then access all the properties of the Event object.

The following example passes the Event object to the `submitForm()` handler function and specifies it as type MouseEvent:

```
<?xml version="1.0"?>
<!-- events/MouseEventHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;

        private function myEventHandler(event:MouseEvent):void {
            // Do something with the MouseEvent object.
            Alert.show("An event of type '" + event.type + "' occurred.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</s:Application>
```

It is best practice to include the `event` keyword when you define all inline event listeners and to specify the most stringent Event object type in the resulting listener function (for example, specify MouseEvent instead of Event).

You can use the Event object to access a reference to the target object (the object that dispatched the event), the type of event (for example, `click`), or other relevant properties, such as the row number and value in a list-based control. You can also use the Event object to access methods and properties of the target component, or the component that dispatched the event.

Although you will most often pass the entire Event object to an event listener, you can just pass individual properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/PropertyHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;

        private function myEventHandler(s:String):void {
            Alert.show("Current Target: " + s);
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me" click="myEventHandler(event.currentTarget.id)"/>
</s:Application>
```

Registering an event listener inline provides less flexibility than using the `addEventListener()` method to register event listeners. The drawbacks are that you cannot set the `useCapture` or `priority` properties on the Event object and that you cannot remove the listener once you add it.

Using the addEventListener() method

The addEventListener() method lets you register event listener functions with the specified control or object. The following example adds the `myClickListener()` function to the b1 instance of a Button control. When the user clicks b1, Flex calls the `myClickListener()` method:

```
b1.addEventListener(MouseEvent.CLICK, myClickListener);
```

The `addEventListener()` method has the following signature:

```
componentInstance.addEventListener(
    event_type:String,
    event_listener:Function,
    use_capture:Boolean,
    priority:int,
    weakRef:Boolean
)
```

The *`event_type`* argument is the kind of event that this component dispatches. This can be either the event type String (for example, “click” or “mouseOut”) or the event type static constant (such as `MouseEvent.CLICK` or `MouseEvent.MOUSE_OUT`). This argument is required.

The constants provide an easy way to refer to specific event types. You should use these constants instead of the strings that they represent. If you misspell a constant name in your code, the compiler catches the mistake. If you instead use strings and make a typographical error, it can be harder to debug and could lead to unexpected behavior.

You should use the constants wherever possible. For example, when you are testing to see whether an Event object is of a certain type, use the following code:

```
if (myEventObject.type == MouseEvent.CLICK) {/* your code here */}
```

Do not use the following code:

```
if (myEventObject.type == "click") {/* your code here */}
```

The *event_listener* argument is the function that handles the event. This argument is required.

The *use_capture* parameter of the addEventListener() method lets you control the phase in the event flow in which your listener will be active. It sets the value of the useCapture property of the Event object. If useCapture is set to true, your listener is active during the capturing phase of the event flow. If useCapture is set to false, your listener is active during the targeting and bubbling phases of the event flow, but not during the capturing phase. The default value is determined by the type of event, but is false in most cases.

To listen for an event during all phases of the event flow, you must call addEventListener() twice, once with the useCapture parameter set to true, and again with *use_capture* set to false. This argument is optional. For more information, see "Capturing phase" on page 82.

The *priority* parameter sets the priority for that event listener. The higher the number, the sooner that event handler executes relative to other event listeners for the same event. Event listeners with the same priority are executed in the order that they were added. This parameter sets the priority property of the Event object. The default value is 0, but you can set it to negative or positive integer values. If several event listeners are added without priorities, the earlier a listener is added, the sooner it is executed. For more information on setting priorities, see "Event priorities" on page 88.

The *weakRef* parameter provides you with some control over memory resources for listeners. A strong reference (when weakRef is false) prevents the listener from being garbage collected. A weak reference (when weakRef is true) does not. The default value is false.

When you add a listener function and that function is invoked, Flex implicitly creates an Event object for you and passes it to the listener function. You must declare the Event object in the signature of your listener function.

If you add an event listener by using the addEventListener() method, you are required to declare an event object as a parameter of the *listener_function*, as the following example shows:

```
b1.addEventListener(MouseEvent.CLICK, performAction);
```

In the listener function, you declare the Event object as a parameter, as follows:

```
public function performAction(e:MouseEvent):void {
    ...
}
```

The following example defines a new handler function myClickListener(). It then registers the click event of the Button control with that handler. When the user clicks the button, Flex calls the myClickHandler() function.

```
<?xml version="1.0"?>
<!-- events/AddEventListenerExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="createListener()">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function createListener():void {
                b1.addEventListener(MouseEvent.CLICK, myClickHandler, false, 0);
            }
            private function myClickHandler(e:MouseEvent):void {
                Alert.show("The button was clicked.");
            }
        ]]>
    </fx:Script>
    <s:Button label="Click Me" id="b1"/>
</s:Application>
```

Using addEventListener() inside an MXML tag

You can add event listeners with the `addEventListener()` method inline with the component definition. The following Button control definition adds the call to the `addEventListener()` method inline with the Button control's `initialize` property:

```
<?xml version="1.0"?>
<!-- events/CallingAddEventListenerInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function myClickHandler(event:Event):void {
               Alert.show("The button was clicked.");
            }
        ]]>
    </fx:Script>
    <s:Button id='b1'
        label="Click Me"
        initialize='b1.addEventListener(MouseEvent.CLICK, myClickHandler, false, 1);'
    />
</s:Application>
```

This is the equivalent of defining the event handler inline. However, defining a handler by using the `addEventListener()` method rather than setting `click="`*`handler_function`*`"` lets you set the value of the `useCapture` and `priority` properties of the Event object. Furthermore, you cannot remove a handler added inline, but when you use the `addEventListener()` method to add a handler, you can call the `removeEventListener()` method to remove that handler.

Using nested inner functions as event listeners

Rather than passing the name of an event listener function to the `addEventListener()` method, you can define an inner function (also known as a closure).

In the following example, the nested inner function is called when the button is clicked:

```
<?xml version="1.0"?>
<!-- events/AddingInnerFunctionListener.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function initApp():void {
            b1.addEventListener("click",
                function(e:Event):void {
                    Alert.show("The button was clicked.");
                }
            );
        }
        ]]>
    </fx:Script>
    <s:Button id='b1' label="Click Me"/>
</s:Application>
```

Function closures are created any time a function is executed apart from an object or a class. They retain the scope in which they were defined. This creates interesting results when a function is passed as an argument or a return value into a different scope.

For example, the following code creates two functions: `foo()`, which returns a nested function named `rectArea()` that calculates the area of a rectangle, and `bar()`, which calls `foo()` and stores the returned function closure in a variable named `myProduct`. Even though the `bar()` function defines its own local variable `x` (with a value of 2), when the function closure `myProduct()` is called, it retains the variable `x` (with a value of 40) defined in function `foo()`. The `bar()` function therefore returns the product of the numbers in the TextInput controls, rather than 8.

```
<?xml version="1.0"?>
<!-- events/FunctionReturnsFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="foo()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        [Bindable]
        private var answer:String;
        private function foo():Function {
            var x:int = int(ti1.text);
            function rectArea(y:int):int { // function closure defined
                return x * y;
            }
            return rectArea;
        }
        private function bar():void {
            var x:int = 2; // ignored
            var y:int = 4; // ignored
            var myProduct:Function = foo();
            answer = myProduct(int(ti2.text)); // function closure called
        }

        ]]>
    </fx:Script>
    <s:Form width="107">
        <s:FormItem label="X">
            <s:TextInput id="ti1" text="10" width="37" textAlign="right"/>
        </s:FormItem>
        <s:FormItem label="Y" width="71">
            <s:TextInput id="ti2" text="20" width="38" textAlign="right"/>
        </s:FormItem>
        <s:Label id="label1" text="{answer}" width="71" textAlign="right"/>
    </s:Form>
    <s:Button id='b1' label="Compute Product" click="bar()"/>

</s:Application>
```

If the listener that you pass to `addEventListener()` method is a nested inner function, you should not pass `true` for the `useWeakReference` argument. For example:

```
addEventListener("anyEvent",
    function(e:Event) { /* My listener function. */ },
    false, 0, true);
```

In this example, passing `true` as the last argument can lead to unexpected results. To Flex, an inner function is actually an object, and can be freed by the garbage collector. If you set the value of the `useWeakReference` argument to `true`, as shown in the previous example, there are no persistent references at all to the inner function. The next time the garbage collector runs, it might free the function, and the function will not be called when the event is triggered.

If there are other references to the inner function (for example, if you saved it in another variable), the garbage collector will not free it.

Regular class-level member functions are not subject to garbage collection; as a result, you can set the value of the `useWeakReference` argument to `true` and they will not be garbage collected.

Removing event handlers

It is a good idea to remove any handlers that will no longer be used. This removes references to objects so that they can be targeted for garbage collection. You can use the `removeEventListener()` method to remove an event handler that you no longer need. All components that can call `addEventListener()` can also call the `removeEventListener()` method. The syntax for the `removeEventListener()` method is as follows:

```
componentInstance.removeEventListener(event_type:String, listener_function:Function,
use_capture:Boolean)
```

For example, consider the following code:

```
myButton.removeEventListener(MouseEvent.CLICK, myClickHandler);
```

The *event_type* and *listener_function* parameters are required. These are the same as the required parameters for the `addEventListener()` method.

The *use_capture* parameter is also identical to the parameter used in the addEventListener() method. Recall that you can listen for events during all event phases by calling addEventListener() twice: once with *use_capture* set to true, and again with it set to false. To remove both event listeners, you must call removeEventListener() twice: once with *use_capture* set to true, and again with it set to false.

You can remove only event listeners that you added with the addEventListener() method in an ActionScript block. You cannot remove an event listener that was defined in the MXML tag, even if it was registered using a call to the addEventListener() method that was made inside a tag attribute.

The following sample application shows what type of handler can be removed and what type cannot:

```
<?xml version="1.0"?>
<!-- events/RemoveEventListenerExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="createHandler(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function createHandler(e:Event):void {
            b1.addEventListener(MouseEvent.CLICK, myClickHandler);
        }
        private function removeMyHandlers(e:Event):void {
            /* Remove listener for b1's click event because it was added
            with the addEventListener() method. */
            b1.removeEventListener(MouseEvent.CLICK, myClickHandler);
            /* Does NOT remove the listener for b2's click event because it
            was added inline in an MXML tag. */
            b2.removeEventListener(MouseEvent.CLICK, myClickHandler);
        }
        private function myClickHandler(e:Event):void {
            Alert.show("The button was clicked.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"/>
    <s:Button label="Click Me Too" id="b2" click="myClickHandler(event)"/>
    <s:Button label="Remove Event Listeners" id="b3" click="removeMyHandlers(event)"/>
</s:Application>
```

Creating event handler classes

You can create an external class file and use the methods of this class as event handlers. Objects themselves cannot be event handlers, but methods of an object can be. By defining one class that handles all your event handlers, you can use the same event handling logic across applications, which can make your MXML applications more readable and maintainable.

To create a class that handles events, you usually import the flash.events.Event class. You also usually write an empty constructor. The following ActionScript class file calls the Alert control's show() method whenever it handles an event with the handleAllEvents() method:

```
// events/MyEventHandler.as
package { // Empty package.
    import flash.events.Event;
    import mx.controls.Alert;
    public class MyEventHandler {
        public function MyEventHandler() {
            // Empty constructor.
        }
        public function handleAllEvents(event:Event):void {
            Alert.show("Some event happened.");
        }
    }
}
```

In your MXML file, you declare a new instance of MyEventHandler and use the `addEventListener()` method to register its `handleAllEvents()` method as a handler to the Button control's `click` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/CustomHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="createHandler()">
    <fx:Script>
        <![CDATA[
            private var myListener:MyEventHandler = new MyEventHandler();
            private function createHandler():void {
                b1.addEventListener(MouseEvent.CLICK, myListener.handleAllEvents);
            }
        ]]>
    </fx:Script>
    <s:Button label="Submit" id="b1"/>
</s:Application>
```

The best approach is to define the event handler's method as static. When you make the event handler method static, you are not required to instantiate the class inside your MXML application. The following `createHandler()` function registers the `handleAllEvents()` method as an event handler without instantiating the MyStaticEventHandler class:

```
<?xml version="1.0"?>
<!-- events/CustomHandlerStatic.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="createHandler()">
    <fx:Script>
        <![CDATA[
            private function createHandler():void {
                b1.addEventListener(MouseEvent.CLICK, MyStaticEventHandler.handleAllEvents);
            }
        ]]>
    </fx:Script>
    <s:Button label="Submit" id="b1"/>
</s:Application>
```

In the class file, you just add the `static` keyword to the method signature:

```
// events/MyStaticEventHandler.as
package { // Empty package.
    import flash.events.Event;
    import mx.controls.Alert;
    public class MyStaticEventHandler {
        public function MyStaticEventHandler() {
            // Empty constructor.
        }
        public static function handleAllEvents(event:Event):void {
            Alert.show("Some event happened.");
        }
    }
}
```

Store your event listener class in a directory in your source path. You can also store your ActionScript class in the same directory as your MXML file, although Adobe does not recommend this.

Defining multiple listeners for a single event

You can define multiple event handler functions for a single event in two ways. When defining events inside MXML tags, you separate each new handler function with a semicolon. The following example adds the `submitForm()` and `debugMessage()` functions as handlers of the `click` event:

```
<?xml version="1.0"?>
<!-- events/MultipleEventHandlersInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        [Bindable]
        private var s:String = "";

        private function submitForm(e:Event):void {
            // Handle event here.
            s += "The submitForm() method was called. ";
        }
        private function debugMessage(e:Event):void {
            // Handle event here.
            s += "The debugMessage() method was called. ";
        }
    ]]></fx:Script>

    <s:Button id="b1"
        label="Do Both Actions"
        click='submitForm(event); debugMessage(event);'
    />
    <s:Label id="l1" text="{s}"/>

    <s:Button id="b2" label="Reset" click="s='';"/>

</s:Application>
```

For events added with the addEventListener() method, you can add any number of handlers with additional calls to the addEventListener() method. Each call adds a handler function that you want to register to the specified object. The following example registers the submitForm() and debugMessage() handler functions with b1's click event:

```
<?xml version="1.0"?>
<!-- events/MultipleEventHandlersAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="createHandlers(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        [Bindable]
        private var s:String = "";
        public function createHandlers(e:Event):void {
            b1.addEventListener(MouseEvent.CLICK, submitForm);
            b1.addEventListener(MouseEvent.CLICK, debugMessage);
        }
        private function submitForm(e:Event):void {
            // Handle event here.
            s += "The submitForm() method was called. ";

        }
        private function debugMessage(e:Event):void {
            // Handle event here.
            s += "The debugMessage() method was called. ";
        }
    ]]></fx:Script>
    <s:Button id="b1" label="Do Both Actions"/>
    <s:Label id="l1" text="{s}"/>

    <s:Button id="b2" label="Reset" click="s='';"/>
</s:Application>
```

You can mix the methods of adding event handlers to any component; alternatively, you can add handlers inline and with the addEventListener() method. The following example adds a click event handler inline for the Button control, which calls the performAction() method. It then conditionally adds a second click handler to call the logAction() method, depending on the state of the CheckBox control.

```
<?xml version="1.0"?>
<!-- events/ConditionalHandlers.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initApp(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function initApp(e:Event):void {
               cb1.addEventListener(MouseEvent.CLICK, handleCheckBoxChange);
               b1.addEventListener(MouseEvent.CLICK, logAction);
            }
            private function handleCheckBoxChange(e:Event):void {
               if (cb1.selected) {
                    b1.addEventListener(MouseEvent.CLICK, logAction);
                    ta1.text += "Added log listener." + "\n";
                } else {
                    b1.removeEventListener(MouseEvent.CLICK, logAction);
                    ta1.text += "Removed log listener." + "\n";
                }
            }
            private function performAction(e:Event):void {
                Alert.show("You performed the action.");
            }
            private function logAction(e:Event):void {
                ta1.text += "Action performed: " + e.type + ".\n";
            }
        ]]>
    </fx:Script>
    <s:Button label="Perform Action" id="b1" click="performAction(event)"/>
    <s:CheckBox id="cb1" label="Log?" selected="true"/>
    <s:TextArea id="ta1" height="200" width="300"/>
</s:Application>
```

You can set the order in which event listeners are called by using the `priority` parameter of the `addEventListener()` method. You cannot set a priority for a listener function if you added the event listener using MXML inline. For more information on setting priorities, see “Event priorities” on page 88.

Registering a single listener with multiple components

You can register the same listener function with any number of events of the same component, or events of different components. The following example registers a single listener function, `submitForm()`, with two different buttons:

```
<?xml version="1.0"?>
<!-- events/OneHandlerTwoComponentsInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function submitForm(e:Event):void {
            // Handle event here.
            Alert.show("Current Target: " + e.currentTarget.id);
        }
        ]]>
    </fx:Script>
    <s:Button id="b1"
        label="Click Me"
        click="submitForm(event)"/>
    <s:Button id="b2"
        label="Click Me, Too"
        click="submitForm(event)"/>

</s:Application>
```

When you use the `addEventListener()` method to register a single listener to handle the events of multiple components, you must use a separate call to the `addEventListener()` method for each instance, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/OneHandlerTwoComponentsAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="createHandlers(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        public function createHandlers(e:Event):void {
            b1.addEventListener(MouseEvent.CLICK, submitForm);
            b2.addEventListener(MouseEvent.CLICK, submitForm);
        }
        private function submitForm(e:Event):void {
            // Handle event here.
            Alert.show("Current Target: " + e.currentTarget.id);
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"/>
    <s:Button id="b2" label="Click Me, Too"/>
</s:Application>
```

When doing this, you should add logic to the event listener that processes the type of event. The event target (or object that dispatched the event) is added to the Event object for you. No matter what triggered the event, you can conditionalize the event processing based on the `target` or `type` properties of the Event object. Flex adds these two properties to all Event objects.

The following example registers a single listener function (`myEventHandler()`) to the `click` event of a Button control and the `click` event of a CheckBox control. To detect what type of object called the event listener, the listener checks the `className` property of the target in the Event object in a `case` statement.

```
<?xml version="1.0"?>
<!-- events/ConditionalTargetHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;

        public function initApp():void {
            button1.addEventListener(MouseEvent.CLICK, myEventHandler);
            cb1.addEventListener(MouseEvent.CLICK, myEventHandler);
        }
        public function myEventHandler(event:Event):void {
            switch (event.currentTarget.className) {
                case "Button":
                    // Process Button click.
                    Alert.show("You clicked the Button control.");
                    break;
                case "CheckBox":
                    // Process CheckBox click.
                    Alert.show("You clicked the CheckBox control.");
                    break;
            }
        }
        ]]>
    </fx:Script>
    <s:Button label="Click Me" id="button1"/>
    <s:CheckBox label="Select Me" id="cb1"/>
</s:Application>
```

Passing additional parameters to listener functions

You can pass additional parameters to listener functions depending on how you add the listeners. If you add a listener with the `addEventListener()` method, you cannot pass any additional parameters to the listener function by arbitrarily adding new parameters to the function signature. The default listener function can declare only a single argument, the Event object (or one of its subclasses).

For example, the following code throws an error because the `clickListener()` method expects two arguments:

```
<fx:Script>
    public function addListeners():void {
        b1.addEventListener(MouseEvent.CLICK,clickListener);
    }
    public function clickListener(e:MouseEvent, a:String):void { ... }
</fx:Script>
<mx:Button id="b1"/>
```

Because the second parameter of the `addEventListener()` method is a function, you can define that function and pass the event object plus any additional parameters through to a different handler. The following example creates a new function in the `addEventListener()` method, add two parameters to the new handler's call, and then handles all of the parameters in the `myClickListener()` method.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- events/CustomListenerFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private var specialParam1:String;
            private var specialParam2:String = "42";
            private function initApp(e:Event):void {
                assignSpecialParam(e);

                /* Change the value of specialParam whenever the user changes it
                   in the TextInput and clicks the button. */
                ti1.addEventListener("focusOut", assignSpecialParam);

                /* Define the pass-through method in the addEventListener() method call.
                   You can add any number of parameters, as long as teh target method's
                   signature agrees. */
                myButton.addEventListener(MouseEvent.CLICK, function (e:MouseEvent):void {
                        myClickListener(e, specialParam1, specialParam2);
                    }
                );
            }

            private function assignSpecialParam(e:Event):void {
                specialParam1 = ti1.text;
            }

            /* This method acts as the event listener, and it has any
             number of parameters that we defined in the addEventListener() call. */
            private function myClickListener(e:MouseEvent, s1:String, s2:String) : void {
                myButton.label = s1 + " " + s2;
            }
        ]]>
    </fx:Script>
    <s:Button id="myButton" label="Click Me"/>
    <s:TextInput id="ti1" text="Enter a custom String here." width="250"/>
</s:Application>
```

Another approach to passing additional parameters to the listener function is to define them in the listener function and then call the final method with those parameters. If you define an event listener inline (inside the MXML tag), you can add any number of parameters as long as the listener function's signature agrees with that number of parameters.

The following example passes a string and the Event object to the `runMove()` method:

```
<?xml version="1.0"?>
<!-- events/MultipleHandlerParametersInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        public function runMove(dir:String, e:Event):void {
            if (dir == "up") {
                moveableButton.y = moveableButton.y - 5;
            } else if (dir == "down") {
                moveableButton.y = moveableButton.y + 5;
            } else if (dir == "left") {
                moveableButton.x = moveableButton.x - 5;
            } else if (dir == "right") {
                moveableButton.x = moveableButton.x + 5;
            }
        }
        ]]>
    </fx:Script>
  <mx:Canvas height="100%" width="100%">
     <s:Button id="moveableButton"
        label="{moveableButton.x.toString()},{moveableButton.y.toString()}"
        x="75"
        y="100"
        width="80"
     />
  </mx:Canvas>
  <s:VGroup horizontalAlign="center">
```

```
     <s:Button id="b1"
        label="Up"
        click='runMove("up",event);'
        width="75"/>
        <s:HGroup horizontalAlign="center">
           <mx:Button id="b2"
            label="Left"
            click='runMove("left",event);'
            width="75"/>
           <s:Button id="b3"
            label="Right"
            click='runMove("right",event);'
            width="75"/>
        </s:HGroup>
     <s:Button id="b4"
        label="Down"
        click='runMove("down",event);'
        width="75"/>
  </s:VGroup>
</s:Application>
```

Manually dispatching events

You can manually dispatch events using a component instance's dispatchEvent() method. All components that extend UIComponent have this method. The method is inherited from the EventDispatcher class, which UIComponent extends.

The syntax for the `dispatchEvent()` method is as follows:

```
objectInstance.dispatchEvent(event:Event):Boolean
```

When dispatching an event, you must create a new Event object. The syntax for the Event object constructor is as follows:

```
Event(event_type:String, bubbles:Boolean, cancelable:Boolean)
```

The *event_type* parameter is the `type` property of the Event object. The *bubbles* and *cancelable* parameters are optional and both default to `false`. For information on bubbling and capturing, see "Event propagation" on page 81.

You can use the `dispatchEvent()` method to dispatch any event you want, not just a custom event. You can dispatch a Button control's `click` event, even though the user did not click a Button control, as in the following example:

```
<?xml version="1.0"?>
<!-- events/DispatchEventExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="createListener(event);">

    <fx:Script><![CDATA[
        import mx.controls.Alert;
        private function createListener(e:Event):void {
            b1.addEventListener(MouseEvent.MOUSE_OVER, myEventHandler);
            b1.addEventListener(MouseEvent.CLICK, myClickHandler);
        }
        private function myEventHandler(e:Event):void {
      var result:Boolean = b1.dispatchEvent(new MouseEvent(MouseEvent.CLICK, true, false));
        }
        private function myClickHandler(e:Event):void {
           Alert.show("The event dispatched by the MOUSE_OVER was of type '" + e.type + "'.");
        }
    ]]></fx:Script>
    <s:Button id="b1" label="Click Me"/>
</s:Application>
```

You can also manually dispatch an event in an MXML tag. In the following example, moving the mouse pointer over the button triggers the button’s `click` event:

```
<?xml version="1.0"?>
<!-- events/DispatchEventExampleInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="createListener(event);">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function createListener(e:Event):void {
            b1.addEventListener(MouseEvent.CLICK, myClickHandler);
        }
        private function myClickHandler(e:Event):void {
           Alert.show("The event dispatched by the MOUSE_OVER was of type '" + e.type + "'.");
        }
        ]]>
    </fx:Script>
    <s:Button id="b1"
        label="Click Me"
        mouseOver="b1.dispatchEvent(new MouseEvent(MouseEvent.CLICK, true, false));"
    />
</s:Application>
```

Your application is not required to handle the newly dispatched event. If you trigger an event that has no listeners, Flex ignores the event.

You can set properties of the Event object in ActionScript, but you cannot add new properties because the object is not dynamic. The following example intercepts a `click` event. It then creates a new MouseEvent object and dispatches it as a `doubleClick` event. In addition, it sets the value of the `shiftKey` property of the MouseEvent object to `true`, to simulate a Shift-click on the keyboard.

```
<?xml version="1.0"?>
<!-- events/DispatchCustomizedEvent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="addListeners()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function customLogEvent(e:MouseEvent):void {
            l1.text = String(e.currentTarget.id);
            l2.text = String(e.type);
            l3.text = String(e.shiftKey);
            // Remove current listener to avoid recursion.
            e.currentTarget.removeEventListener("doubleClick", customLogEvent);
        }
        private function handleEvent(e:MouseEvent):void {
            // Add new handler for custom event about to be dispatched.
            e.currentTarget.addEventListener("doubleClick", customLogEvent);
            // Create new event object.
            var mev:MouseEvent = new MouseEvent("doubleClick");
            // Customize event object.
            mev.shiftKey = true;
            // Dispatch custom event.
            e.currentTarget.dispatchEvent(mev);
        }
        private function addListeners():void {
            b1.addEventListener("click",handleEvent);
            b2.addEventListener("click",handleEvent);
        }
    ]]></fx:Script>
        <s:Button id="b1" label="Click Me (b1)"/>
        <s:Button id="b2" label="Click Me (b2)"/>

        <s:Form>
            <s:FormItem label="Current Target:">
                <s:Label id="l1"/>
            </s:FormItem>
            <s:FormItem label="Event Type:">
                <s:Label id="l2"/>
            </s:FormItem>
            <s:FormItem label="Shift Key Pressed:">
                <s:Label id="l3"/>
            </s:FormItem>
        </s:Form>
</s:Application>
```

If you want to add custom properties to an Event object, you must extend the Event object and define the new properties in your own custom class. You can then manually dispatch your custom events with the `dispatchEvent()` method, as you would any event.

If you create a custom ActionScript class that dispatches its own events but does not extend UIComponent, you can extend the flash.events.EventDispatcher class to get access to the `addEventListener()`, `removeEventListener()`, and `dispatchEvent()` methods.

To make your code more efficient, you can check to see if the intended target of a dispatched event is listening for the event. If it is not, then there is no reason to dispatch the event. This is especially true if the event bubbles because that means many calls will be made by the system while it searches for a target. You can check by using the `hasEventListner()` and `willTrigger()` methods. Peter deHaan has a blog post on using these methods.

Event propagation

When events are triggered, there are three phases in which Flex checks whether there are event listeners. These phases occur in the following order:

- Capturing
- Targeting
- Bubbling

During each of these phases, the nodes have a chance to react to the event. For example, assume the user clicks a Button control that is inside a VBox container. During the capturing phase, Flex checks the Application object and the VBox for listeners to handle the event. Flex then triggers the Button's listeners in the target phase. In the bubbling phase, the VBox and then the Application are again given a chance to handle the event but now in the reverse order from the order in which they were checked in the capturing phase.

In ActionScript 3.0, you can register event listeners on a target node and on any node along the event flow. Not all events, however, participate in all three phases of the event flow. Some types of events are dispatched directly to the target node and participate in neither the capturing nor the bubbling phases. All events can be captured unless they are dispatched from the top node.

Other events may target objects that are not on the display list, such as events dispatched to an instance of the Socket class. These event objects flow directly to the target node, without participating in the capturing or bubbling phases. You can also cancel an event as it flows through the event model so that even though it was supposed to continue to the other phases, you stopped it from doing so. You can do this only if the `cancelable` property is set to `true`.

Capturing and bubbling happen as the Event object moves from node to node in the display list: parent-to-child for capturing and child-to-parent for bubbling. This process has nothing to do with the inheritance hierarchy. Only DisplayObject objects (visual objects such as containers and controls) can have a capturing phase and a bubbling phase in addition to the targeting phase.

Mouse events and keyboard events are among those that bubble. Any event can be captured, but no DisplayObject objects listen during the capturing phase unless you explicitly instruct them to do so. In other words, capturing is disabled by default.

When a faceless event dispatcher, such as a Validator, dispatches an event, there is only a targeting phase, because there is no visual display list for the Event object to capture or bubble through.

About the target and currentTarget properties

Every Event object has a `target` and a `currentTarget` property that help you to keep track of where it is in the process of propagation. The `target` property refers to the dispatcher of the event. The `currentTarget` property refers to the current node that is being examined for event listeners.

When you handle a mouse event such as `MouseEvent.CLICK` by writing a listener on some component, the `event.target` property does not necessarily refer to that component; it is often a subcomponent, such as the Button control's UITextField, that defines the label.

When Flash Player or Adobe® AIR™ dispatches an event, it dispatches the event from the frontmost object under the mouse. Because children are in front of parents, that means the player or AIR might dispatch the event from an internal subcomponent, such as the UITextField of a Button.

The `event.target` property is set to the object that dispatched the event (in this case, UITextField), not the object that is being listened to (in most cases, you have a Button control listen for a `click` event).

MouseEvent events bubble up the parent chain, and can be handled on any ancestor. As the event bubbles, the value of the `event.target` property stays the same (UITextField), but the value of the `event.currentTarget` property is set at each level to be the ancestor that is handling the event. Eventually, the `currentTarget` will be Button, at which time the Button control's event listener will handle the event. For this reason, you should use the `event.currentTarget` property rather than the `event.target` property; for example:

```
<mx:Button label="OK" click="trace(event.currentTarget.label)"/>
```

In this case, in the Button event's click event listener, the `event.currentTarget` property always refers to the Button, while `event.target` might be either the Button or its UITextField, depending on where on the Button control the user clicked.

Capturing phase

In the capturing phase, Flex examines an event target's ancestors in the display list to see which ones are registered as a listener for the event. Flex starts with the root ancestor and continues down the display list to the direct ancestor of the target. In most cases, the root ancestors are the Stage, then the SystemManager, and then the Application object.

For example, if you have an application with a Panel container that contains a TitleWindow container, which in turn contains a Button control, the structure appears as follows:

```
Application
    Panel
        TitleWindow
            Button
```

If your listener is on the `click` event of the Button control, the following steps occur during the capturing phase if capturing is enabled:

1 Check the Application container for click event listeners.

2 Check the Panel container for click event listeners.

3 Check the TitleWindow container for click event listeners.

During the capturing phase, Flex changes the value of the `currentTarget` property on the Event object to match the current node whose listener is being called. The `target` property continues to refer to the dispatcher of the event.

By default, no container listens during the capturing phase. The default value of the *use_capture* argument is `false`. The only way to add a listener during this phase is to pass `true` for the *use_capture* argument when calling the `addEventListener()` method, as the following example shows:

```
myPanel.addEventListener(MouseEvent.MOUSE_DOWN, clickHandler, true);
```

If you add an event listener inline with MXML, Flex sets this argument to `false`; you cannot override it.

If you set the *use_capture* argument to true—in other words, if an event is propagated through the capturing phase—the event can still bubble, but capture phase listeners will not react to it. If you want your event to traverse both the capturing and bubbling phases, you must call addEventListener() twice: once with *use_capture* set to true, and then again with *use_capture* set to false.

The capturing phase is very rarely used, and it can also be computationally intensive. By contrast, bubbling is much more common.

Targeting phase

In the targeting phase, Flex invokes the event dispatcher's listeners. No other nodes on the display list are examined for event listeners. The values of the currentTarget and the target properties on the Event object during the targeting phase are the same.

Bubbling phase

In the bubbling phase, Flex examines an event's ancestors for event listeners. Flex starts with the dispatcher's immediate ancestor and continues up the display list to the root ancestor. This is the reverse of the capturing phase.

For example, if you have an application with a Panel container that contains a TitleWindow container that contains a Button control, the structure appears as follows:

```
Application
    Panel
        TitleWindow
            Button
```

If your listener is on the click event of the Button control, the following steps occur during the bubble phase if bubbling is enabled:

1 Check the TitleWindow container for click event listeners.

2 Check the Panel container for click event listeners.

3 Check the Application container for click event listeners.

An event only bubbles if its bubbles property is set to true. Mouse events and keyboard events are among those that bubble; it is less common for higher-level events that are dispatched by Flex to bubble. Events that can be bubbled include change, click, doubleClick, keyDown, keyUp, mouseDown, and mouseUp. To determine whether an event bubbles, see the event's entry in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

During the bubbling phase, Flex changes the value of the currentTarget property on the Event object to match the current node whose listener is being called. The target property continues to refer to the dispatcher of the event.

When Flex invokes an event listener, the Event object might have actually been dispatched by an object deeper in the display list. The object that originally dispatched the event is the target. The object that the event is currently bubbling through is the currentTarget. So, you should generally use the currentTarget property instead of the target property when referring to the current object in your event listeners.

If you set the useCapture property to true—in other words, if an event is propagated through the capturing phase—then it does not bubble, regardless of its default bubbling behavior. If you want your event to traverse both the capturing and bubbling phases, you must call addEventListener() twice: once with useCapture set to true, and then again with useCapture set to false.

An event only bubbles up the parent's chain of ancestors in the display list. Siblings, such as two Button controls inside the same container, do not intercept each other's events.

Detecting the event phase

You can determine what phase you are in by using the Event object's `eventPhase` property. This property contains an integer that represents one of the following constants:

- 1 — Capturing phase (`CAPTURING_PHASE`)
- 2 — Targeting phase (`AT_TARGET`)
- 3 — Bubbling phase (`BUBBLING_PHASE`)

The following example displays the current phase and information about the current target:

```
<?xml version="1.0"?>
<!-- events/DisplayCurrentTargetInfo.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script><![CDATA[
        import mx.controls.Alert;
        private function showInfo(e:MouseEvent):void {
            Alert.show("Phase: " + e.eventPhase + "\n" +
                "ID: " + e.currentTarget.id + "\n" +
                "Label: " + e.currentTarget.label + "\n" +
                "Font Size: " + e.currentTarget.getStyle("fontSize"), "Current Target Info");
        }
    ]]></fx:Script>
    <s:Button id="b1" label="Click Me" click="showInfo(event)"/>
</s:Application>
```

Stopping propagation

During any phase, you can stop the traversal of the display list by calling one of the following methods on the Event object:

- stopPropagation()
- stopImmediatePropagation()

You can call either the event's `stopPropagation()` method or the `stopImmediatePropagation()` method to prevent an Event object from continuing on its way through the event flow. The two methods are nearly identical and differ only in whether the current node's remaining event listeners are allowed to execute. The `stopPropagation()` method prevents the Event object from moving on to the next node, but only after any other event listeners on the current node are allowed to execute.

The `stopImmediatePropagation()` method also prevents the Event objects from moving on to the next node, but it does not allow any other event listeners on the current node to execute.

The following example creates a TitleWindow container inside a Panel container. Both containers are registered to listen for a `mouseDown` event. As a result, if you click on the TitleWindow container, the `showAlert()` method is called twice unless you add a call to the `stopImmediatePropagation()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/StoppingPropagation.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="init(event);">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        import flash.events.MouseEvent;
        import flash.events.Event;
        public function init(e:Event):void {
            p1.addEventListener(MouseEvent.MOUSE_DOWN, showAlert);
            tw1.addEventListener(MouseEvent.MOUSE_DOWN, showAlert);
            tw1.addEventListener(Event.CLOSE, closeWindow);
            p2.addEventListener(MouseEvent.MOUSE_DOWN, showAlertWithoutStoppingPropagation);
            tw2.addEventListener(MouseEvent.MOUSE_DOWN, showAlertWithoutStoppingPropagation);
            tw2.addEventListener(Event.CLOSE, closeWindow);
        }
        public function showAlert(e:Event):void {
            Alert.show("Alert!\n" + "Current Target: " + e.currentTarget + "\n" +
                "Phase: " + e.eventPhase);
            e.stopImmediatePropagation();
        }
        public function showAlertWithoutStoppingPropagation(e:Event):void {
            Alert.show("Alert!\n" + "Current Target: " + e.currentTarget + "\n" +
                "Phase: " + e.eventPhase);
        }
        public function closeWindow(e:Event):void {
            p1.removeChild(tw1);
        }
        ]]>
    </fx:Script>
    <s:Panel id="p1" title="Stops Propagation">
```

```
        <mx:TitleWindow id="tw1"
            width="300"
            height="100"
            showCloseButton="true"
            title="Title Window 1">
            <s:Button label="Click Me"/>
            <s:TextArea id="ta1"/>
        </mx:TitleWindow>
    </s:Panel>
    <s:Panel id="p2" title="Does Not Stop Propagation">
        <mx:TitleWindow id="tw2"
            width="300"
            height="100"
            showCloseButton="true"
            title="Title Window 2">
            <s:Button label="Click Me"/>
            <s:TextArea id="ta2"/>
        </mx:TitleWindow>
    </s:Panel>
</s:Application>
```

***Note:** A call to either the `Event.stopPropogation()` or the `Event.stopImmediatePropogation()` methods does not prevent default behavior from occurring.*

Event examples

In the following example, the parent container’s click handler disables the target control after the target handles the event. It shows that you can reuse the logic of a single listener (click the HGroup container) for multiple events (all the clicks).

```
<?xml version="1.0"?>
<!-- events/NestedHandlers.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        public function disableControl(event:MouseEvent):void {
            // Use this same logic for all events.
            event.currentTarget.enabled = false;
        }
        public function doSomething(event:MouseEvent):void {
            b1.label = "clicked";
            ta1.text += "Something happened.";
        }
        public function doSomethingElse(event:MouseEvent):void {
            b2.label = "clicked";
            ta1.text += "Something happened again.";
        }
    ]]></fx:Script>
    <s:HGroup id="hb1" height="200" click="disableControl(event)">
        <s:Button id='b1' label="Click Me" click="doSomething(event)"/>
        <s:Button id='b2' label="Click Me" click="doSomethingElse(event)"/>
        <s:TextArea id="ta1"/>
    </s:HGroup>
    <s:Button id="resetButton"
        label="Reset"
        click="hb1.enabled=true;b1.enabled=true;b2.enabled=true;b1.label='Click 
Me';b2.label='Click Me';"/>
</s:Application>
```

By having a single listener on a parent control instead of many listeners (one on each child control), you can reduce your code size and make your applications more efficient. Reducing the number of calls to the `addEventListener()` method potentially reduces application startup time and memory usage.

The following example registers an event handler for the Panel container, rather than registering a listener for each link. All children of the Panel container inherit this event handler. Since Flex invokes the handler on a bubbled event, you use the `target` property rather than the `currentTarget` property. In this handler, the `currentTarget` property would refer to the Panel control, whereas the `target` property refers to the LinkButton control, which has the label that you want.

```
<?xml version="1.0"?>
<!-- events/SingleRegisterHandler.mxml -->
<s:Application     xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="createLinkHandler();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        private function linkHandler(event:MouseEvent):void {
            try {
                var url:URLRequest = new URLRequest("http://finance.google.com/finance?q=" +
                event.target.label);
                navigateToURL(url);
            } catch (e:Error) {
                /**
                *  Do nothing; just want to catch the error that occurs when a user clicks on
                 *  the Panel and not one of the LinkButtons.
                 **/
            }
        }
        private function createLinkHandler():void {
            p1.addEventListener(MouseEvent.CLICK,linkHandler);
        }
        ]]>
    </fx:Script>

    <s:Panel id="p1" title="Click on a stock ticker symbol">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <mx:LinkButton label="ADBE"/>
        <mx:LinkButton label="GE"/>
        <mx:LinkButton label="IBM"/>
        <mx:LinkButton label="INTC"/>
    </s:Panel>
</s:Application>
```

Event priorities

You can register any number of event listeners with a single event. Flex registers event listeners in the order in which the addEventListener() methods are called. Flex then typically calls the listener functions when the event occurs in the order in which they were registered. However, if you register some event listeners inline and some with the `addEventListener()` method, the order in which the listeners are called for a single event can be unpredictable.

You can change the order in which Flex calls event listeners by using the *priority* parameter of the `addEventListener()` method. It is the fourth argument of the `addEventListener()` method.

Flex calls event listeners in priority order, from highest to lowest. The highest priority event is called first. In the following example, Flex calls the `verifyInputData()` method before the `saveInputData()` function. The `verifyInputData()` method has the highest priority. The last method to be called is `returnResult()` because the value of its `priority` parameter is lowest.

```
<?xml version="1.0"?>
<!-- events/ShowEventPriorities.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        private function returnResult(e:Event):void {
            ta1.text += "returnResult() method called last (priority 1)\n";
        }
        private function verifyInputData(e:Event):void {
            ta1.text += "verifyInputData() method called first (priority 3)\n";
        }
        private function saveInputData(e:Event):void {
            ta1.text += "saveInputData() method called second (priority 2)\n";
        }
        private function initApp():void {
            b1.addEventListener(MouseEvent.CLICK, returnResult, false, 1);
            b1.addEventListener(MouseEvent.CLICK, saveInputData, false, 2);
            b1.addEventListener(MouseEvent.CLICK, verifyInputData, false, 3);
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"/>

    <s:TextArea id="ta1" height="200" width="300"/>

</s:Application>
```

You can set the event priority to any valid integer, positive or negative. The default value is 0. If multiple listeners have the same priority, Flex typically calls them in the order in which they were registered, although the order is not guaranteed.

If you want to change the priority of an event listener once the event listener has already been defined, you must remove the listener by calling the `removeEventListener()` method. You add the event again with the new priority.

The *priority* parameter of the `addEventListener()` method is not an official part of the DOM Level 3 events model. ActionScript 3.0 provides it so that programmers can be more flexible when organizing their event listeners.

If your listeners rely on a specific order of execution, you can call one listener function from within another, or dispatch a new event from within the first event listener. For more information on manually dispatching events, see "Manually dispatching events" on page 78.

Using event subclasses

Depending on the event type, the Event object can have a wide range of properties. These properties are based on those defined in the W3C specification http://www.w3.org/TR/DOM-Level-3-Events/), but Flex does not implement all of these.

When you declare an Event object in a listener function, you can declare it of type Event, or you can specify a subclass of the Event object. In the following example, you specify the event object as type MouseEvent:

```
public function performAction(e:MouseEvent):void {
    ...
}
```

Most controls generate an object that is of a specific event type; for example, a mouse click generates an object of type MouseEvent. By specifying a more specific event type, you can access specific properties without having to cast the Event object to something else. In addition, some subclasses of the Event object have methods that are unique to them. For example, the LogEvent has a `getLevelString()` method, which returns the log level as a String. The generic Event object does not have this method.

An event object that you define at run time can be a subclass of the compile-time type. You can access the event-specific properties inside an event listener even if you did not declare the specific event type, as long as you cast the Event object to a specific type. In the following example, the function defines the object type as Event. However, inside the function, in order to access the `localX` and `localY` properties, which are specific to the MouseEvent class, you must cast the Event object to be of type MouseEvent.

```
<?xml version="1.0"?>
<!-- events/AccessEventSpecificProperties.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="addListeners()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function customLogEvent(e:Event):void {
                var a:MouseEvent = MouseEvent(e);
                Alert.show("Y: " + a.localY + "\n" + "X: " + a.localX);
            }
            private function addListeners():void {
                b1.addEventListener(MouseEvent.CLICK, customLogEvent);
            }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"/>
</s:Application>
```

If you declare the Event object as a specific type, you are not required to cast that object in the handler, as the following example shows:

```
private function customLogEvent(e:MouseEvent):void { ... }
```

In the previous example, you can also cast the Event object for only the property access, using the syntax shown in the following example:

```
<?xml version="1.0"?>
<!-- events/SinglePropertyAccess.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="addListeners()">
    <fx:Script>
        <![CDATA[
        import mx.controls.Alert;
        private function customLogEvent(e:Event):void {
            Alert.show("Y: " + MouseEvent(e).localY + "\n" + "X: " + MouseEvent(e).localX);
        }
        private function addListeners():void {
            b1.addEventListener(MouseEvent.CLICK, customLogEvent);
        }
        ]]>
    </fx:Script>

    <s:Button id="b1" label="Click Me"/>
</s:Application>
```

This approach can use less memory and system resources, but it is best to declare the event’s type as specifically as possible.

Each of the Event object’s subclasses provides additional properties and event types that are unique to that category of events. The MouseEvent class defines several event types related to that input device, including the `CLICK`, `DOUBLE_CLICK`, `MOUSE_DOWN`, and `MOUSE_UP` event types.

For a list of types for each Event subclass, see the subclass’s entry in *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About keyboard events

It is common for applications to respond to a key or series of keys and perform some action—for example, Control+q to quit the application. While Flash Player supports all the basic functionality of key combinations from the underlying operating system, it also lets you override or trap any key or combination of keys to perform a custom action.

Handling keyboard events

In some cases, you want to trap keys globally, meaning no matter where the user is in the application, their keystrokes are recognized by the application and the action is performed. Flex recognizes global keyboard events whether the user is hovering over a button or the focus is inside a TextInput control.

A common way to handle global key presses is to create a listener for the `KeyboardEvent.KEY_DOWN` or `KeyboardEvent.KEY_UP` event on the application. Listeners on the application container are triggered every time a key is pressed, regardless of where the focus is (as long as the focus is in the application on not in the browser controls or outside of the browser). Inside the handler, you can examine the key code or the character code using the `charCode` and `keyCode` properties of the KeyboardEvent class, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/TrapAllKeys.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.core.FlexGlobals;

        private function initApp():void {
          FlexGlobals.topLevelApplication.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
        }
        private function keyHandler(event:KeyboardEvent):void {
            t1.text = event.keyCode + "/" + event.charCode;
        }
        ]]>
    </fx:Script>
    <s:TextInput id="myTextInput"/>

    <s:Label id="t1"/>
</s:Application>
```

To run this example, you must first set the focus to something inside the application, such as the TextInput control, by clicking on it.

Because any class that extends UIComponent dispatches the `keyUp` and `keyDown` events, you can also trap keys pressed when the focus is on an individual component.

Understanding the keyCode and charCode properties

You can access the `keyCode` and `charCode` properties to determine what key was pressed and trigger other actions as a result. The `keyCode` property is a numeric value that corresponds to the value of a key on the keyboard. The `charCode` property is the numeric value of that key in the current character set (the default character set is UTF-8, which supports ASCII). The primary difference between the key code and character values is that a key code value represents a particular key on the keyboard (the 1 on a keypad is different than the 1 in the top row, but the 1 on the keyboard and the key that generates the ! are the same key), and the character value represents a particular character (the *R* and *r* characters are different).

The mappings between keys and key codes are device and operating system dependent. ASCII values, on the other hand, are available in the ActionScript documentation.

The following example shows the character and key code values for the keys you press. When you run this example, you must be sure to put the focus in the application before beginning.

```
<?xml version="1.0"?>
<!-- charts/ShowCharAndKeyCodes.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init()"
    width="650">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
     import flash.events.KeyboardEvent;
     private function init():void {
        ti1.setFocus();
        this.addEventListener(KeyboardEvent.KEY_DOWN, trapKeys);
     }

     private function trapKeys(e:KeyboardEvent):void {
        ta1.text = String(e.toString());

        l1.text = numToChar(e.charCode) + " (" + String(e.charCode) + ")";
        l2.text = numToChar(e.keyCode) + " (" + String(e.keyCode) + ")";
     }

    private function numToChar(num:int):String {
        if (num > 47 && num < 58) {
            var strNums:String = "0123456789";
            return strNums.charAt(num - 48);
        } else if (num > 64 && num < 91) {
            var strCaps:String = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
            return strCaps.charAt(num - 65);
        } else if (num > 96 && num < 123) {
            var strLow:String = "abcdefghijklmnopqrstuvwxyz";
            return strLow.charAt(num - 97);
        } else {
            return num.toString();
        }
    }
  ]]></fx:Script>
  <s:TextInput width="50%" id="ti1"/>

  <s:Panel id="mainPanel" width="100%" height="100%">
     <s:Form>
        <s:FormItem label="Char (Code)">
           <s:Label id="l1"/>
        </s:FormItem>
        <s:FormItem label="Key (Code)">
           <s:Label id="l2"/>
        </s:FormItem>
        <s:FormItem label="Key Event">
           <s:TextArea id="ta1" width="500" height="200" editable="false"/>
        </s:FormItem>
     </s:Form>
  </s:Panel>

</s:Application>
```

You can listen for specific keys or combinations of keys by using a conditional operator in the KeyboardEvent handler. The following example listens for the combination of the Shift key plus the q key and prompts the user to close the browser window if they press those keys at the same time:

```
<?xml version="1.0"?>
<!-- events/TrapQKey.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.core.FlexGlobals;

        private function initApp():void {
          FlexGlobals.topLevelApplication.addEventListener(KeyboardEvent.KEY_UP,keyHandler);

            // Set the focus somewhere inside the application.
            ta1.setFocus();
        }
        //This function quits the application if the user presses Shift+Q.
        private function keyHandler(event:KeyboardEvent):void {
            var bShiftPressed:Boolean = event.shiftKey;
            if (bShiftPressed) {
                var curKeyCode:int = event.keyCode;
                if (curKeyCode == 81) { // 81 is the keycode value for the Q key
                    /* Quit the application by closing the browser using JavaScript.
                       This may not work in all browsers. */
                    var url:URLRequest = new
                    URLRequest("javascript:window.close()");
                    navigateToURL(url,"_self");
                }
            }
        }
        ]]>
    </fx:Script>
    <s:TextArea id="ta1" text="Focus here so that Shift+Q will quit the browser."/>
</s:Application>
```

Notice that this application must have focus when you run it in a browser so that the application can capture keyboard events.

Understanding KeyboardEvent precedence

If you define `keyUp` or `keyDown` event listeners for both a control and its parent, you will notice that the keyboard event is dispatched for each component because the event bubbles. The only difference is that the `currentTarget` property of the KeyboardEvent object is changed.

In the following example, the application, the `my_vgroup` container, and the `my_textinput` control all dispatch `keyUp` events to the `keyHandler()` event listener function:

```
<?xml version="1.0"?>
<!-- events/KeyboardEventPrecedence.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();"
    width="650">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import mx.core.FlexGlobals;

        private function initApp():void {
          FlexGlobals.topLevelApplication.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
            my_vgroup.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
            my_textinput.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
            // Set the focus somewhere inside the application.
            my_textinput.setFocus();
        }
        private function keyHandler(event:KeyboardEvent):void {
            ta1.text += event.target + "(" + event.currentTarget + "): " +
                event.keyCode + "/" + event.charCode + "\n";
        }
    ]]></fx:Script>

    <s:VGroup id="my_vgroup">
        <s:TextInput id="my_textinput"/>
    </s:VGroup>
    <s:TextArea id="ta1" height="300" width="550"/>
</s:Application>
```

When you examine the output, you will notice that the `target` property of the KeyboardEvent object stays the same because it refers to the original dispatcher of the event (in this case, my_textinput). But the `currentTarget` property changes depending on what the current node is during the bubbling (in this case, it changes from my_textinput to my_vgroup to the application itself).

The order of calls to the event listener is determined by the object hierarchy and not the order in which the `addEventListener()` methods were called. Child controls dispatch events before their parents. In this example, for each key pressed, the TextInput control dispatches the event first, the VGroup container next, and finally the application.

When handling a key or key combination that the underlying operating system or browser recognizes, the operating system or browser generally processes the event first. For example, in Microsoft Internet Explorer, pressing Control+w closes the browser window. If you trap that combination in your application, Internet Explorer users never know it, because the browser closes before the ActiveX Flash Player has a chance to react to the event.

Handling keyboard-related mouse events

The MouseEvent class and all MouseEvent subclasses (such as ChartItemEvent, DragEvent, and LegendMouseEvent) have the following properties that you can use to determine if a specific key was held down when the event occurred:

Property	Description
`altKey`	Is set to `true` if the Alt key was held down when the user pressed the mouse button; otherwise, `false`.
`ctrlKey`	Is set to `true` if the Control key was held down when the user pressed mouse button; otherwise, `false`.
`shiftKey`	Is set to `true` if the Shift key was held down when the user pressed mouse button; otherwise, `false`.

The following example deletes Button controls, based on whether the user holds down the Shift key while pressing the mouse button:

```
<?xml version="1.0"?>
<!-- events/DetectingShiftClicks.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import spark.components.Button;
        private function initApp():void {
            var b1:Button = new Button();
            b1.label = "Button 1";
            var b2:Button = new Button();
            b2.label = "Button 2";
            b1.addEventListener(MouseEvent.CLICK, removeButtons);
            b2.addEventListener(MouseEvent.CLICK, removeButtons);
            vg1.addElement(b1);
            vg1.addElement(b2);
        }
        private function removeButtons(event:MouseEvent):void {
            if (event.shiftKey) {
                vg1.removeElement(Button(event.currentTarget));
            } else {
                event.currentTarget.toolTip = "Shift+click to remove this button.";
            }
        }
    ]]></fx:Script>
    <s:VGroup id="vg1"/>

    <s:Button id="resetButton" label="Reset" click="initApp();"/>
</s:Application>
```

SDK configuration

Use the configuration files included with Adobe® Flex® SDK to configure the compilers and other aspects of Flex.

About configuration files

Root variables

For Flex SDK, the *flex_install_dir* variable is the top-level directory where you installed the SDK. Under this directory are the bin, frameworks, lib, and samples directories. The *flex_app_root* directory is the top level location for many files.

Configuration files layout

The layout of the configuration files for Flex SDK is simple. It includes a jvm.config file, fdb command-line debugger shell script, and the mxmlc and compc command-line compiler shell scripts for configuring the JVM that the compiler uses. It also includes the flex-config.xml file that sets the compiler options, as well as executable files for fdb, mxmlc, and compc.

The layout of the configuration files for Flex SDK is as follows:

```
sdk_install_dir/
        bin/jvm.config
        bin/mxmlc
        bin/mxmlc.exe
        bin/compc
        bin/compc.exe
        bin/fdb
        bin/fdb.exe
        frameworks/flex-config.xml
```

Flex SDK configuration

Flex SDK includes the mxmlc and compc command-line compilers. You use mxmlc to compile applications from MXML, ActionScript, and other source files. You use the compc compiler to compile component libraries, Runtime Shared Libraries (RSLs), and theme files.

The compilers are located in the *sdk_install_dir*/bin directory. You can configure the compiler options with the flex-config.xml file or command line options.

The compilers use the Java JRE. As a result, you can also configure settings such as memory allocation and source path with the JVM arguments.

***Note:** You must use a 32-bit version of the JDK, version 1.5 or later. The command line compilers do not work with a 64-bit JDK.*

Command-line compiler configuration

The flex-config.xml file defines the default compiler options for the compc and mxmlc command-line compilers. You can use this file to set options such as debugging, SWF file metadata, and themes to apply to your application. For a complete list of compiler options, see “Using mxmlc, the application compiler” on page 2174 and “Using compc, the component compiler” on page 2194.

The flex-config.xml file is located in the *sdk_install_dir*/frameworks directory. If you change the location of this file relative to the location of the command-line compilers, you can use the `load-config` compiler option to point to its new location.

You can also use a local configuration file that overrides the compiler options of the flex-config.xml file. You give this local configuration file the same name as the MXML file, plus "-config.xml" and store it in the same directory. For example, MyApp-config.xml. When you compile your MXML file, the compiler looks for a local configuration file first, then the flex-config.xml file.

For more information on compiler configuration files, see "About configuration files" on page 2171.

JVM configuration

The Flex compilers use the Java JRE. Configuring the JVM can result in faster and more efficient compilations. Without a JVM, you cannot use the mxmlc and compc command-line compilers. You can configure JVM settings such as the Java source path, Java library path, and memory settings.

You must use JDK 1.5 or later with the command-line compilers. These compilers are 32-bit executables and cannot launch 64-bit processes. As a result, they require a 32-bit version of Java to run. If you have both a 32-bit and 64-bit JDK installed, you can use several techniques to target the 32-bit version for compilation, including:

- Using the `JAVA_HOME` environment variable to point to the 32-bit version of the JDK when you compile with mxmlc or compc
- Modifying the Flex SDK's jvm.config file to point to the 32-bit version of the JDK
- Using batch files that point to the 32-bit version of the JDK to launch the compilers

On Windows, you use the compc.exe and mxmlc.exe executable files in the bin directory to compile Flex applications and component libraries. You use the fdb.exe executable file in the bin directory to debug applications. The executable files use the jvm.config file to set JVM arguments. The jvm.config file is in the same directory as the executable files. If you move it or the executable files to another directory, they use their default settings and not the settings defined in the jvm.config file.

The fdb, compc, and mxmlc shell scripts (for UNIX, Linux, or Windows systems running a UNIX-shell emulator such as Cygwin) do not take a configuration file. You set the JVM arguments inside the shell script file.

The jvm.config file is used by the Java process opened by the mxmlc and compc command-line executable files. The file is located in *sdk_install_dir*/bin.

Changing the JVM heap size

The most common JVM configuration is to set the size of the Java heap. The Java heap is the amount of memory reserved for the JVM. The actual size of the heap during run time varies as classes are loaded and unloaded. If the heap requires more memory than the maximum amount allocated, performance will suffer as the JVM performs garbage collection to maintain enough free memory for the applications to run.

You can set the initial heap size (or minimum) and the maximum heap size on most JVMs. By providing a larger heap size, you give the JVM more memory with which to defer garbage collection. However, you must not assign all of the system's memory to the Java heap so that other processes can run optimally.

To set the initial heap size on the Sun HotSpot JVM, change the value of the `Xms` property. To change the maximum heap size, change the value of the `Xmx` property. The following example sets the initial heap size to 256M and the maximum heap size to 512M:

```
java.args=-Xms256m -Xmx512m -Dsun.io.useCanonCaches=false
```

In addition to increasing your JVM's heap size, you can tune the JVM in other ways. Some JVMs provide more granular control over garbage collecting, threading, and logging. For more information, consult your JVM documentation or view the options on the command line. If you are using the Sun HotSpot JVM, for example, you can enter `java -X` or `java -D` on the command line to see a list of configuration options.

In many cases, you can also use a different JVM. Benchmark your Flex application and the application server on several different JVMs. Choose the JVM that provides you with the best performance.

Setting the `useCanonCaches` argument to `false` is required to support Windows file names.

Flash Player configuration

You can use the standard version or the debugger version of Adobe® Flash® Player as clients for your Flex applications. The debugger version of Flash Player can log output from the `trace()` global method as well as data services messages and custom log events.

You enable and disable logging and configure the location of the output file in the mm.cfg file. For more information on locating and editing the mm.cfg file, see "Editing the mm.cfg file" on page 2224.

You can configure the standard version and the debugger version of Flash Player to use auto-update and other settings by using the mms.cfg file. This file is in the same directory as the mm.cfg file. For more information on auto-update, see the Flash Player documentation.

Chapter 3: Application architecture

Application development phases

Design phase

In the design phase, you make basic decisions about how to write code for reusability, how your application interacts with its environment, how your application accesses application resources, and many other decisions. In the design phase, also define your development and deployment environments, including the directory structure of your application.

Although these design decisions specify how your application interacts with its environment, you also have architectural issues to decide. For example, you might choose to develop your application based on a particular design pattern, such as Model-View-Controller (MVC).

About design patterns

One common starting point of the design phase is to identify one or more design patterns relevant for your application. A design pattern describes a solution to a common programming problem or scenario. Although the design pattern might give you insight into how to approach an application design, it does not necessarily define how to write code for that solution.

Many types of design patterns have been catalogued and documented. For example, the Functional design pattern specifies that each module of your application performs a single action, with little or no side effects for the other modules in your application. The design pattern does not specify what a module is, commonly though it corresponds to a class or method.

About MVC

The goal of the Model-View-Controller (MVC) architecture is that by creating components with a well-defined and limited scope in your application, you increase the reusability of the components and improve the maintainability of the overall system. Using the MVC architecture, you can partition your system into three categories of components:

Model components Encapsulates data and behaviors related to the data processed by the application. The model might represent an address, the contents of a shopping cart, or the description of a product.

View components Defines your application's user interface, and the user's view of application data. The view might contain a form for entering an address, a DataGrid control for showing the contents of a shopping cart, or an image of a product.

Controller components Handles data interconnectivity in your application. The Controller provides application management and the business logic of the application. The Controller does not necessarily have any knowledge of the View or the Model.

For example, with the MVC design, you could implement a data-entry form that has three distinct pieces:

- The model consists of XML data files or the remote data service calls to hold the form data.
- The view is the presentation of any data and display of all user interface elements.
- The controller contains logic that manipulates the model and sends the model to the view.

The promise of the MVC architecture is that by creating components with a well-defined and limited scope, you increase the reusability of these components and improve the maintainability of the overall system. In addition, you can modify components without affecting the entire system.

Although you can consider a Flex application as part of the View in a distributed MVC architecture, you can use Flex to implement the entire MVC architecture on the client. A Flex application has its own view components that define the user interface, model components that represent data, and controller components that communicate with back-end systems.

About Struts

Struts is an open-source framework that facilitates the development of web applications based on Java servlets and other related technologies. Because it provides a solution to many of the common problems that developers face when building these applications, Struts has been widely adopted in a large variety of development efforts, from small projects to large-scale enterprise applications.

Struts is based on a Model-View-Controller (MVC) architecture, with a focus on the controller part of the MVC architecture. In addition, it provides JSP tag libraries to help you create the view in a traditional JSP/HTML environment.

Configure phase

Before you write your first line of application code, or before you deploy an application, you must ensure that you configure your environment correctly. Configuration is a broad term and encompasses several different tasks.

For example, you must configure your development and deployment environments to ensure that your application can access the required resources and data services. If your application requires access to a web service, ensure that your application has the correct access rights to the web service. If your application runs outside a firewall, ensure that it can access resources inside the firewall.

The following sections contain an overview of configuration tasks.

About run-time configuration

Most run-time configuration has to do with configuring access to remote data services, such as web services. For example, during application development, you run your application behind a firewall, where the application has access to all necessary resources and data services. However, when you deploy the application, you must ensure that an executing application can still access the necessary resources when the application runs outside of the firewall.

One configuration issue for Flex SDK applications is the placement of a crossdomain.xml file. For security, by default Flash Player does not allow an application to access a remote data service from a domain other than the domain from which the application was served. Therefore, a server that hosts a data service must be in the same domain as the server hosting your application, or the remote server must define a crossdomain.xml file. A crossdomain.xml file is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from specific domains, or from all domains. By default, place the crossdomain.xml at the root directory of the server that is serving the data.

Flex SDK does not include a server-side proxy for handling data service requests. Therefore, you must ensure that you configure data services for direct access by your application, or make data service requests through your own proxy server.

Build phase

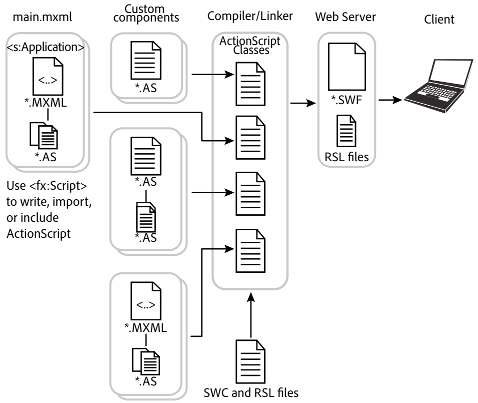
Building your application is an iterative process that includes three main tasks:

1. Compile
2. Debug
3. Test

About compiling

Compiling your application converts your application files and assets into a single SWF file. During compilation, you set compiler options to enable accessibility, enable debug information in the output, set library paths, and set other options. You can configure the compiler as part of configuring your project in Flash Builder, by using command-line arguments to the compiler, or by setting options in a configuration file.

When you compile your application, the Flex compiler creates a single SWF file from all of the application files (Adobe® MXML™, AS, RSL, SWC, and asset files), as the following example shows:



Flex provides two compilers: mxmlc and compc. You can use the compc and mxmlc compilers from within Flash Builder or from a command line.

You use mxmlc to compile MXML, ActionScript, SWC, and RSL files into a single SWF file. After your application is compiled and deployed on your web or application server, a user can make an HTTP request to download and play the SWF file on their computer.

You use compc to create resources that you use to create the application. For example, you can compile components, classes, and other files into SWC files or into RSLs, and then statically or dynamically link these libraries to your application.

For more information, see “Flex compilers” on page 2164.

About debugging an application

Flex provides several tools that you use to debug your application, including the following:

AIR Debug Launcher (ADL) A command line version of the Adobe® AIR™ debugger that you can use outside of Adobe® Flex™ Builder™.

Flash Player You can run Flex applications in two different versions of Adobe® Flash® Player: the standard version, which the general public uses, and the debugger version, which application developers use to debug their applications during the development process.

Flash Builder visual debugger The Flash Builder debugger allows you to run and debug applications. You can use the debugger to set and manage breakpoints; control application execution by suspending, resuming, and terminating the application; step into and over the code; watch variables; evaluate expressions; and so on.

Flex Command-line debugger A command line version of the debugger that you can use outside of Flash Builder.

For more information, see “Command-line debugger” on page 2209.

About testing an application

Due to the size, complexity, and large amounts of data handled by applications, maintaining the quality of a large software application can be difficult. To help with this task, you can use automated testing tools that test and validate application behavior without human intervention.

The Flex Automation Package provides developers with the ability to create Flex applications that use the Automation API. You can use this API to create automation agents or to ensure that your applications are ready for testing. In addition, the Flex Automation Package includes support for HP QuickTest Professional (QTP) automation tool. For more information, see “Creating applications for testing” on page 2270.

Deploy phase

When you deploy your application, you make it available to customers. Typically, you deploy the application as a SWF file on a web server so that users can access it by using an HTTP request to the SWF file.

When you deploy the application’s SWF file, you must also deploy all of the assets required by the application. For example, if the application requires access to video or image files, or to XML data files, you must make sure to deploy those assets as well. If the application uses an RSL, you must also deploy the RSL.

Deploying assets may not necessarily be as simple as copying the assets to a location on your web server. Flash Player has built-in security features that controls the access of application assets at run time.

This section contains an overview of the deployment phase. For more information, see “Deploying applications” on page 2544.

What happens during a request to a SWF file

When a customer requests the SWF file, the web server or application server returns the SWF file to the client computer. The SWF file then runs locally on the client.

In some cases, a request to a Flex SWF file can cause multiple requests to multiple SWF files. For example, if your application uses Runtime Shared Libraries (RSLs), the web server or application server returns an RSL as a SWC file to the client along with the application SWF file.

Server-side caching

Your web server or application server typically caches the SWF file on the first request, and then serves the cached file on subsequent requests. You configure server-side caching by using the options available in your web server or application server.

Client-side caching

The SWF file returned to the client is typically cached by the customer’s browser on first request. Depending on the browser configuration, the SWF file typically remains in the cache until the browser closes. When the browser reopens, the next request to the SWF file must reload it from the server.

Integrating Flex applications with your web application

To incorporate a Flex application into a website, you typically embed the SWF file in an HTML, JSP, Adobe® ColdFusion®, or other type of web page. The page that embeds the SWF file is known as the *wrapper*.

A wrapper consists of an `<object>` tag and an `<embed>` tag that format the SWF file on the page, define data object locations, and pass run-time variables to the SWF file. In addition, the wrapper can include support for deep linking and Flash Player version detection and deployment.

When you compile an application with Flash Builder, it automatically creates a wrapper file for you in the bin directory associated with the Flash Builder project. You can copy the contents of the wrapper file into your HTML pages to reference the SWF file. Flash Builder uses the SWFObject 2 library to embed the SWF file in the HTML page.

You can edit the wrapper to manipulate how Flex appears in the browser. You can also add JavaScript or other logic in the page to communicate with Flex or generate customized pages.

When using the mxmlc command-line compiler, you generally write the wrapper yourself. You can also use the wrapper included with Flex as a template for creating your own. If you use ant to compile your applications, you can use the `html-wrapper` ant task.

More Help topics

"Creating a wrapper" on page 2552

"Using the html-wrapper task" on page 2350

Secure phase

Security is not necessarily a phase of the application development process, but is an issue that you should take into consideration during the entire development process. That is, you do not configure, build, test, and deploy an application, and then define the security issues. Rather, you take security into consideration during all phases.

Building security into your application often takes the following main efforts:

- Using the security features built into Flash Player
- Building security into your application

Flash Player has several security features built into it, including sandbox security, that you can take advantage of because you are building applications for Flash Player.

But, Flash Player security is not enough for many application requirements. For example, your application may require the user to log in, or perform authentication in some other way, before accessing data services. When you must handle security issues beyond those built into Flash Player, design them into your application from the initial design phase, test them during the compile phase, and verify them during the deploy phase.

For more information on security, see "Security" on page 117.

About the security model

The Flex security model protects both the client and the server. Consider the following general aspects of security when you deploy Flex applications:

- Flash Player operating in a sandbox on the client
- Authorizing and authenticating users who access a server's resources

  Flash Player runs inside a security sandbox that prevents the client from being hijacked by malicious application code. This sandbox prevents a user from running a Flex application that can access system files and perform other tasks.

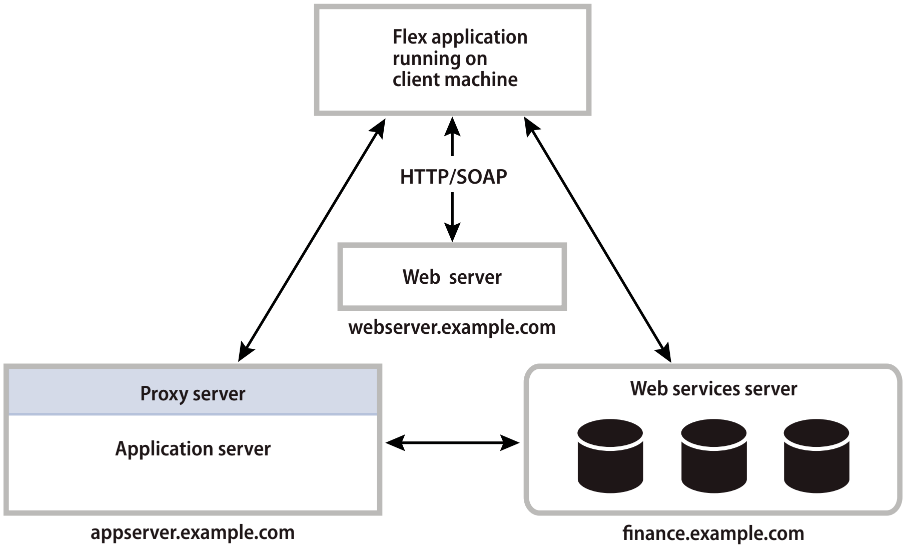
Flash Player security

Flash Player has an extensive list of features that ensure Flash content is secure, including the following:

- Uses the encryption capabilities of SSL in the browser to encrypt all communications between a Flash application and the server
- Includes an extensive sandbox security system that limits transfer of information that might pose a risk to security or privacy
- Does not allow applications to read data from the local drive, except for SharedObjects that were created by that domain
- Does not allow writing any data to the disk except for data that is encapsulated in SharedObjects
- Does not allow web content to read any data from a server that is not from the same domain, unless that server explicitly allows access
- Enables the user to disable the storage of information for any domain
- Does not allow data to be sent from a camera or microphone unless the user gives permission

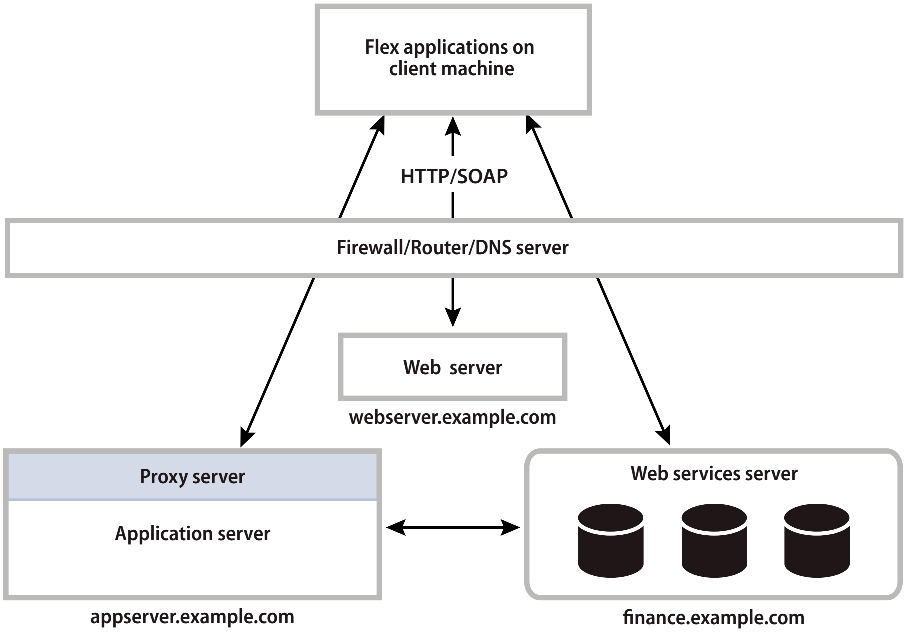
Application Development in Flex SDK

The following example shows a typical development environment for a Flex SDK application:



In this example, application development happens in an environment that is behind a firewall, and you deploy your application SWF file on webserver.example.com. To run the application, you make a request to it from a computer that is also within the firewall. The executing SWF file can access resources on any other server as necessary. In the development environment, the SWF file can directly access web services, or it can access them through a proxy server.

The following example shows a typical deployment environment for a Flex SDK application:



In this example, the customer requests the application SWF file from webserver.example.com, the server returns the SWF file to the customer, and the SWF file plays. The executing SWF file must be able to access the necessary resources from outside the firewall.

Design phase

With Flex SDK, one of your first design decisions might be to choose a design pattern that fits your application requirements. That design pattern might have implications on how you structure your development environment, determine the external data services that your application must access, and define how you integrate your Flex application into a larger web application.

Configure phase

For run-time configuration, you ensure that your executing SWF file can access the necessary resources including asset files (such as image files) and external data services. If you access a resource on a domain other than the domain from which the SWF file is served, you must define a crossdomain.xml file on the target server, or make the request through a proxy server.

Build phase

To build an application for Flex SDK, you define a directory structure on your development system for application files, and define the location of application assets. You then compile, debug, and test your application.

The compile-time configuration for a Flex SDK application is primarily a process of setting compiler options to define the location of SWC and RSLs, to create a SWF file with debug information, or to set additional compiler options. When compiling applications, you compile your application into a single SWF file, and then deploy the SWF file to a web server or application server for testing.

Deploy phase

With Flex SDK, you deploy your application SWF file on your web server or application server. Users then access the deployed SWF file by making an HTTP request in the form:

```
http://hostname/path/filename.swf
```

If you embed your SWF file in an HTML or other type of web page using a wrapper, users request the wrapper page. The request to the wrapper page causes the web server or application server to return the SWF file along with the wrapper page.

Secure phase

Security issues for Flex SDK applications often have to do with how the application accesses external resources. For example, you might require a user to log in to access resources, or you might want the application to be able to access external data services that implement some other form of access control.

Application structure

Installation directory structure

Before you can begin to set up your application development environment, be familiar with the Flex installation directory structure for the following products:

- Flex SDK
- Adobe ® Flash® Builder™

Flex SDK installation directory structure

When you install Flex SDK, the installer creates the following directory structure under the installation directory:

Directory	Description
/ant	Contains the Flex Ant tasks, which provide a convenient way to build your Flex projects.
/asdoc	Contains ASDoc, a command-line tool that you can use to create API language reference documentation as HTML pages from the classes in your Flex application.
/bin	Contains the executable files, such as the mxmlc and compc compilers.
/frameworks	Contains configuration files, such as flex-config.xml and default.css.
/frameworks/libs	Contains the library SWC files. You use the files to compile your application.
/frameworks/locale	Contains the localization resource files.
/frameworks/projects	Contains the Flex framework source code.
/frameworks/rsls	Contains the RSL for the Flex framework.
/frameworks/themes	Contains the theme files that define the basic look and feel of all Flex components.
/lib	Contains JAR files.
/runtimes	Contains the standard and debugger versions of Adobe ® Flash® Player and the Adobe® AIR™ components.
/samples	Contains sample applications.
/templates	Contains template HTML wrapper files. This includes history management files for deep linking as well as the playerProductInstall.swf file for Player updating.

Flash Builder installation directory structure

When you install Flash Builder, you install Flex SDK plus Flash Builder. The installer creates the following directory structure:

Directory	Description
Flash Builder 4.6	The top-level directory for Flash Builder.
/configuration	A standard Eclipse folder that contains the config.ini file and error logs.
/features	A standard Eclipse folder that contains the plug-ins corresponding to features of Flash Builder.
/jre	Contains the Java Runtime Environment installed with Flash Builder used by default when you run the stand-alone version of Flash Builder.
/player	Contains the different versions of Flash Player—the standard version and the debugger version.
/plugins	Contains the Eclipse plugins used by Flash Builder.
/sdks	Contains the different Flex SDKs. For a directory description, see "Flex SDK installation directory structure" on page 107.

Development directory structure

As part of the process of setting up the directory structure of your development environment, you define the directory location for application-specific assets, assets shared across applications, and the location of other application files and assets.

Flex file types

A Flex application consists of many different file types. Consider the following options when you decide where to place each type of file.

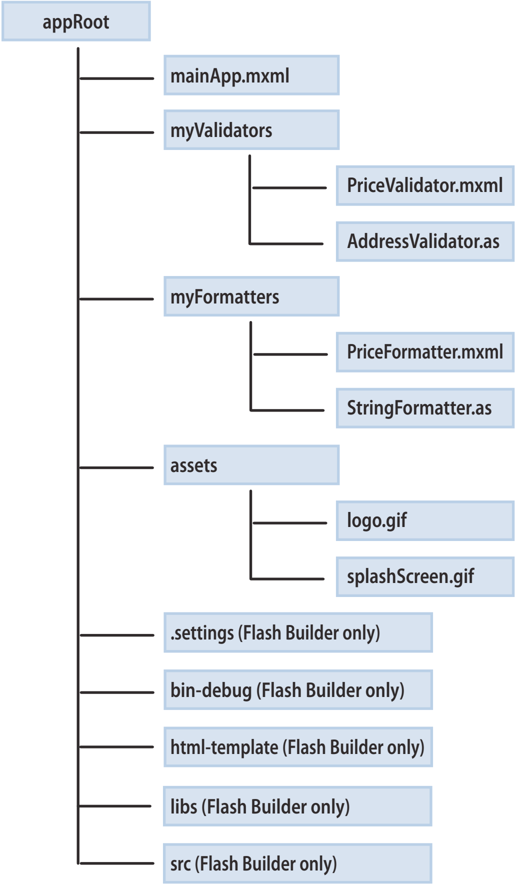
The following table describes the different Flex file types:

File format	Extension	Description
MXML	.mxml	Your application typically has one main application MXML file that contains the `<s:Application>` tag, and one or more MXML files that implement your custom MXML components.
ActionScript	.as	A utility class, Flex custom component class, or other logic implemented as an ActionScript file.
SWC	.swc	A custom library file, or a custom component implemented as an MXML or ActionScript file, then packaged as a SWC file. A SWC file contains components that you package and reuse among multiple applications. The SWC file is then statically linked into your application at compile time when you create the application's SWF file.
RSL	.swc	A custom library implemented as an MXML or ActionScript file, and then deployed as a Runtime Shared Library (RSL). An RSL is a stand-alone SWC file that is downloaded separately from your application's SWF file, cached on the client computer for use with multiple application SWF files, and dynamically linked to your application.
CSS file	.css	A text file template for creating a Cascading Style Sheets file.
Assets	.flv, .mp3, .jpg, .gif, .swf, .png, .svg, .xml, other	The assets required by your application, including image, skin, sound, and video files.

Flex SDK directory structure

A typical Flex application consists of a main MXML file (the file that contains the `<s:Application>` tag), one or more MXML files that implement custom MXML components, one or more ActionScript files that contains custom components and custom logic, and asset files.

The following example shows an example of the directory structure of a simple Flex application:



This application consists of a root application directory and directories for different types of files. Everything required to compile and run the application is contained in the directory structure of the application.

Flash Builder adds additional directories to the application that are not present for Flex SDK applications:

.settings Contains the preference settings for your Flash Builder project

bin-debug Contains the debug SWF and debug wrapper files

bin-release Contains the generated SWF file and wrapper file, created by Flash Builder when you select File > Export > Release Version

html-template Contains additional files used by specific Flex features, such as deep linking or Player detection. Flash Builder uses these files to generate an HTML wrapper for your SWF file.

libs Contains additional SWC files and other files used by the application.

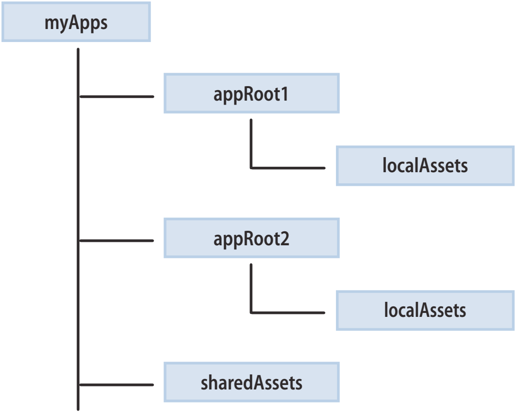
src Contains the application source code.

There are no inherent restrictions in Flex for the location of the root directory of your application, so you can put it almost anywhere in the file system of your computer. If you are using Flash Builder, the default location of the application root directory in Microsoft Windows is My Documents\Flash Builder 4.6*project_name* (for example, C:\Documents and Settings*userName*\My Documents\Flash Builder 4.6\myFlexApp).

Sharing assets among applications

Typically, you do not develop a single application in isolation from all other applications. Your application shares files and assets with other applications.

The following example shows two Flex applications, appRoot1 and appRoot2. Each application has a directory for local assets, and can access shared assets from a directory outside of the application's directory structure:



The location of the shared assets does not have to be at the same level as the root directories of the Flex applications. It only needs to be somewhere accessible by the applications at compile time.

In the following example, you use the Image control in an MXML file in the appRoot1 directory to access an asset from the shared assets directory:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apparch/EmbedExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:Image id="loader1"
        source="@Embed(source='../assets/butterfly.gif')"/>
</s:Application>
```

Consideration for accessing application assets

One of the decisions that you must make when you create a Flex application is whether to load your assets at run time, or to embed the assets within the application's SWF file.

When you embed an asset, you compile it into your application's SWF file. The advantage to embedding an asset is that it is included in the SWF file, and can be accessed faster than having to load it from a remote location at run time. The disadvantage of embedding is that your SWF file is larger than if you load the asset at run time.

If you decide to access an asset at run time, you can load it from the local file system of the computer on which the SWF file runs, or you can access a remote asset, typically though an HTTP request over a network.

A SWF file can access one type of external asset: local or over a network; the SWF file cannot access both types. You determine the type of access allowed by the SWF file by using the `use-network` flag when you compile your application. When you set the `use-network` flag to `false`, you can access assets in the local file system, but not over the network. The default value is `true`, which lets you access assets over the network, but not in the local file system.

For more information on the `use-network` flag, see "Flex compilers" on page 2164. For more information on embedding application assets, see "Embedding assets" on page 1699.

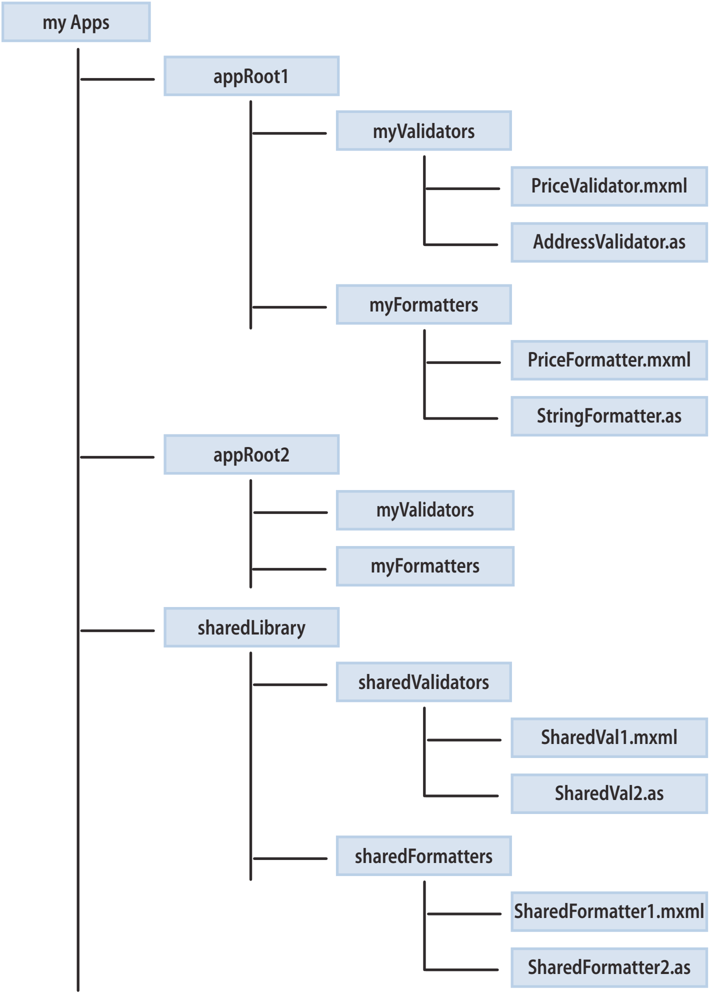
Sharing MXML and ActionScript files among applications

You can build an entire Flex application in a single MXML file that contains both your MXML code and any supporting ActionScript code. As your application gets larger, your single file also grows in size and complexity. This type of application would soon become difficult to understand and debug, and very difficult for multiple developers to work on simultaneously.

Flex supports a component-based development model. You use the predefined components included with Flex to build your applications, and create components for your specific application requirements. You can create custom components using MXML or ActionScript.

Defining your own components has several benefits. One advantage is that components let you divide your applications into modules that you can develop and maintain separately. By implementing commonly used logic within custom components, you can also build a suite of reusable components that you can share among multiple Flex applications.

The following example shows two Flex applications, appRoot1 and appRoot2. Each application has a subdirectory for local MXML and ActionScript components, and can also reference a library of shared components:



The Flex compiler uses the source path to determine the directories where it searches for MXML and ActionScript files. By default, the root directory of the application is included in the source path; therefore, a Flex application can access any MXML and ActionScript files in its main directory, or in a subdirectory.

For shared MXML and ActionScript files that are outside of the application's directory structure, you modify the source path to include the directories that the compiler searches for MXML and ActionScript files. The component search order in the source path is based on the order of the directories listed in the source path.

You can set the source path as part of configuring your project in Flash Builder, in the flex-config.xml file, or set it when you open the command-line compiler. In this example, you set the source path to:

```
C:\myApps\sharedLibrary
```

To access a component in an MXML file, you specify a namespace definition that defines the directory location of the component relative to the source path. In the following example, an MXML file in the appRoot1 directory accesses an MXML component in the local directory structure, and in the directory containing the shared library of components:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apparch/ComponentNamespaces.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyLocalComps="myFormatters.*"
    xmlns:MySharedComps="sharedFormatters.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <MyLocalComps:PriceFormatter/>
    <MySharedComps:SharedFormatter2/>
</s:Application>
```

The MXML tag name for a custom component is composed of two parts: the namespace prefix, in this example MyLocalComps and MySharedComps, and the tag name. The namespace prefix tells Flex the directory in the source path that contains the file that implements the custom component. The tag name corresponds to the filename of the component, in this example PriceFormatter.mxml and SharedFormatter2.mxml.

Using a SWC file in a Flex SDK application

A SWC file is a Flex library file that contains one or more components implemented in MXML or ActionScript. All Flex library files are shipped as SWC files in the frameworks/libs directory. The list of library files includes:

- framework.swc
- rpc.swc
- core.swc
- charts.swc
- advanceddatagrid.swc

You can also create SWC files that you package and reuse among multiple applications. You typically use static linking with SWC files, which means the compiler includes all components, classes, and their dependencies in the application SWF file when you compile the application. For more information on static linking, see “About linking” on page 254.

By default, the Flex compiler includes all linked SWC files in the frameworks/libs directory when it compiles your application. SWC files that contain only classes that are not used in your application are not included. For your custom SWC files, you use the `library-path` option of the mxmlc compiler, or set the library path in Flash Builder, to specify the location of the SWC file.

Using an RSL in a Flex SDK application

One way to reduce the size of your application’s SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but must be transferred to the client only once. These shared files are known as Runtime Shared Libraries or RSLs.

An RSL is a stand-alone file that the client downloads separately from your application’s SWF file, and caches on the client computer for use with multiple application SWF files. Using an RSL reduces the resulting file size for your applications. The benefits increase as the number of applications that use the RSL increases. If you only have one application, putting components into RSLs does not reduce the aggregate download size, and might increase it.

You create an RSL as a SWC file that you package and reuse among multiple applications. To reference an RSL, you use the `runtime-shared-libraries` option for the command-line compiler, or Flash Builder. You typically use dynamic linking with RSLs, which means that the classes in the RSL are left in an external file that is loaded at run time.

Every Flex application uses some aspects of the Flex framework, which is a relatively large set of ActionScript classes that define the infrastructure of a Flex application. If a client loads two different Flex applications, the application will likely load overlapping class definitions. To further reduce the SWF file size, you can use *framework* RSLs. Framework RSLs let you externalize the framework libraries and can be used with any Flex application.

For more information on RSLs, framework RSLs, and dynamic linking, see "Runtime Shared Libraries" on page 253.

Using modules in a Flex SDK application

Modules are SWF files that can be loaded and unloaded by an application. They cannot be run independently of an application, but any number of applications can share the modules.

Modules let you split your application into several separate SWF files. The main application, or shell, can dynamically load other SWF files that it requires, when it needs them. It does not have to load all modules when it starts, nor does it have to load any modules if the user does not interact with them. When the application no longer needs a module, it can unload the module to free up memory and resources.

For more information, see "Modular applications" on page 138.

Compiling an application

Compiling your application converts your application files and assets into a single SWF file. During compilation, you set compiler options to enable accessibility, enable debug information in the output, set library paths, and set other options. You can configure the compiler as part of configuring your project in Flash Builder, by using command-line arguments to the compiler, or by setting options in a configuration file.

For more information on compiling applications, see "Flex compilers" on page 2164.

About case sensitivity during a compile

The Flex compilers use a case-sensitive file lookup on all file systems. On case-insensitive file systems, such as the Macintosh and Windows file systems, the Flex compiler generates a case-mismatch error when you use a component with the incorrect case. On case-sensitive file systems, such as the UNIX file system, the Flex compiler generates a component-not-found error when you use a component with the incorrect case.

Compiling a Flex SDK application

Flex SDK includes two compilers, mxmlc and compc. You use mxmlc to compile MXML files, ActionScript files, SWC files, and RSLs into a single SWF file. After your application is compiled and deployed on your web or application server, a user can make an HTTP request to download and play the SWF file on their computer. You use the compc compiler to compile components, classes, and other files into SWC files or RSLs.

To compile an application with Flex SDK, you use the mxmlc compiler in the bin directory of your Flex SDK directory. The most basic mxmlc example is one in which the MXML file for your application has no external dependencies (such as components in a SWC file or ActionScript classes). In this case, you open mxmlc from the command line and point it to your MXML file, as the following example shows:

```
$ mxmlc c:/myFiles/app.mxml
```

The mxmlc compiler has many options that you can specify on the command line, or that you can set in the flex-config.xml file. For example, to disable warning messages, you set the `warnings` options to `false`, as the following example shows:

```
$ mxmlc -warnings=false c:/myFiles/app.mxml
```

You only specify the main application file, the file that contains the `<s:Application>` tag, to the compiler. The compiler searches the default source path for any MXML and ActionScript files that your application references. If your application references MXML and ActionScript files in directories that are not included in the default source path, you can use the `sourcepath-` option to add a directory to the source path, as the following example shows:

```
$ mxmlc -source-path path1 path2 path3 c:/myFiles/app.mxml
```

In this example, you specify a list of directories, separated by spaces, and terminate that list with `--`.

For more information on mxmlc options, see "Using mxmlc, the application compiler" on page 2174.

Compiling an application that uses SWC files

Often, you use SWC files when compiling MXML files. You specify the SWC files in the compiler by using the `library-path` option.

The following example adds two SWC files to the `library-path` when it compiles your application:

```
$ mxmlc -library-path+=/myLibraries/MyRotateEffect.swc;/myLibraries/MyButtonSwc.swc
c:/myFiles/app.mxml
```

Compiling an application that uses RSLs

To use an RSL in your application, use the `runtime-shared-library-path` compiler option. The following example compiles an application with an RSL at the command line:

```
$ mxmlc -runtime-shared-library-path=../lib/mylib.swc,../bin/library.swf Main.mxml
```

Compiling an application that uses modules

The way you compile modules is similar to the way you compile applications. On the command line, you use the mxmlc command-line compiler. The result is a SWF file that you load into your application as a module.

You cannot run the module-based SWF file as a stand-alone application or load it into a browser window. It must be programmatically loaded by an application as a module.

For more information on compiling modules on the command line, see "Modular applications" on page 138.

Compiling a Flash Builder application

When you compile a project with Flash Builder, you run the Flex compilers from within Flash Builder itself, not from the command line. You can build your projects manually or let Flash Builder automatically compile them for you. In either case, the Flash Builder compiler creates the application files, generates a wrapper, places the output files in the proper location, and alerts you to any errors encountered during compilation. You then run and debug your applications as needed.

For mobile applications, Flash Builder creates and deploys the APK file for you.

If you must modify the default build settings, you have several options for controlling how your projects are built into applications. For example, you can set build preferences on individual projects or on all the projects in your workspace, modify the build output path, change the build order, and so on. You can also create custom build instructions using third-party tools, such as Apache Ant.

When your projects are built, automatically or manually, Flash Builder places the SWF file in the project output folder along with the wrapper. By default, this is the debug version of your application. It contains debugging information and, therefore, is used when you debug your application. A wrapper file embeds the application SWF file and is used to run or debug your application in a web browser. The standard version of your application SWF files, which you generate through Export Release Version, does not include the additional debugging information and is smaller.

For more information about compiling Flash Builder applications, see Build projects.

Compiling an application that uses modules

In Flash Builder, you create modules as applications and compile them by either building the project or running the application. The result is a SWF file that you load into your application as a module.

You cannot run the module-based SWF file as a stand-alone Flex application or load it into a browser window. It must be loaded by an application as a module. When you run it in Flash Builder to compile it, you should close the Player or browser Window and ignore any errors. Modules should not be requested by the Player or through a browser directly.

For information on compiling modules in Flash Builder, see "Modular applications" on page 138.

Deployment directory structure

When you deploy an application, ensure that the directory structure of the deployed application is correct.

When you deploy your application, must be aware of how your application accesses its assets. If you embedded all of your application assets into the SWF file, you can deploy the application as a stand-alone SWF file.

However, if your application accesses assets at run time, the application requests assets during execution. You must ensure that you deploy all of the necessary assets, in the correct location, so that you can run the application correctly.

Assets that you deploy at run time include:

Asset type	More information
HTML wrapper	"Creating a wrapper" on page 2552
Deep linking files	"Deep linking" on page 2022
Express Install files	"Using Express Install in the wrapper" on page 2558
RSLs	"Runtime Shared Libraries" on page 253
Modules	"Modular applications" on page 138
Sub-applications	"Developing and loading sub-applications" on page 176
Compiled CSS SWF files	"Loading style sheets at run time" on page 1547
Resource modules	"Resource Bundles" on page 2091
Images, sound files, and other binary assets that are not embedded	
Data files	

In some cases, the deployed locations of these files must match the locations of the files during development. For example, if you load modules from the same directory as your main application, then you must deploy these modules to that directory, unless you programmatically handle alternative locations to load the modules from.

In other cases, the deployed locations of these files is specified. For example, the deep linking files history.css, historyFrame.html, and history.js must all reside in a /history subdirectory that is located relative to the application's SWF file.

And in other cases, you specify the eventual deployed location of these assets when you compile your application. For example, if you compiled your application using an RSL, you must ensure that the RSL is also deployed to your web server, along with your application's SWF file. The directory location of the RSL must match the directory location that you specified at compile time using the `runtime-shared-libraries` or `runtime-shared-library-path` options for the compiler.

For more information about what assets to deploy with your application, see "Deployment checklist" on page 2548.

Security

Introduction to security

Adobe® Flash® Player runs applications built as SWF files. Content is delivered as a series of instructions in binary format to Flash Player over web protocols in the precisely described SWF (.swf) file format. The SWF files themselves are typically hosted on a server and then downloaded to, and displayed on, the client computer when requested. Most of the content consists of binary ActionScript instructions. ActionScript is the ECMA standards-based scripting language that Flash uses that features APIs designed to allow the creation and manipulation of client-side user interface elements and for working with data.

The Flex security model protects both client and the server. Consider the following two general aspects to security:

- Authorization and authentication of users accessing a server's resources
- Flash Player operating in a sandbox on the client

Flex supports working with the web application security of any J2EE application server. In addition, precompiled applications can integrate with the authentication and authorization scheme of any underlying server technology to prevent users from accessing your applications. The Flex framework also includes several built-in security mechanisms that let you control access to web services, HTTP services, and server-based resources such as EJBs.

Flash Player runs inside a security sandbox that prevents the client from being hijacked by malicious application code.

***Note:** SWF content running in the Adobe® AIR™ follows different security rules than content running in the browser. For details, see Adobe AIR Security.*

Declarative compared to programmatic security

The two common approaches to security are declarative and programmatic. Often, declarative security is server based. Using the server's configuration, you provide protection to a resource or set of resources. You use the container's authentication and authorization schemes to protect that resource from unauthorized access.

The declarative approach to security casts a wide net. Declarative security is implemented as a separate layer from the web components that it works with. You set up a security system, such as a set of file permissions or users, groups, and roles, and then you plug your application's authentication mechanism into that layer.

With declarative security, either a user gains access to the resource or they do not. Usually the content cannot be customized based on roles. In an HTML-based application, the result is that users are denied access to certain pages. However, in a Flex environment, the typical result of declarative security is that the user is denied access to the entire application, since the application is seen as a single resource to the container.

Declarative security lets programmers who write web applications ignore the environment in which they write. Declarative security is typically set up and maintained by the deployer and not the developer of the application. Also, updates to the web application do not generally require a refactoring of the security model.

Programmatic security gives the developer of the application more control over access to the application and its resources. Programmatic security can be much more detailed than declarative security. For example, a developer using programmatic security can allow or deny a user access to a particular component inside the application.

Although programmatic security is typically configured by the developer of the application, it usually interacts with the same systems as declarative security, so the relationship between developer and deployer of the application must be cooperative when implementing programmatic security.

Declarative security is recommended over programmatic security for most applications because the design promotes code reuse, making it more maintainable. Furthermore, declarative security puts the responsibility of security into the hands of the people who specialize in its implementation; application programmers can concentrate on writing applications and people who deploy the applications in a specific environment can concentrate on enforcing security policies and take advantage of that context.

Client security overview

When considering security issues, you cannot think of applications as traditional web applications. Applications built with Flex often consist of a single monolithic SWF file that is loaded by the client once, or a series of SWF files loaded as modules or RSLs. Web applications, on the other hand, usually consist of many individual pages that are loaded one at a time.

Most web applications access resources such as web services that are outside of the client. When an application accesses an external resource, two factors apply:

- Is the user authorized to access this resource?
- Can the client load the resource, or is it prevented from loading the resource, because of its sandbox limitations?

The following basic security rules always apply by default:

- Resources in the same security sandbox can always access each other.
- SWF files in a remote sandbox can never access local files and data.

You should consider the following security issues related to the client architecture that affect applications.

Flash Player security features

Much of Flash Player security is based on the domain of origin for loaded SWF files, media, and other assets. A SWF file from a specific Internet domain, such as www.example.com, can always access all data from that domain. These assets are put in the same security grouping, known as a security sandbox. For example, a SWF file can load SWF files, bitmaps, audio, text files, and any other asset from its own domain. Also, cross-scripting between two SWF files from the same domain is permitted, as long as both files are written using ActionScript 3.0. Cross-scripting is the ability of one SWF file to use ActionScript to access the properties, methods, and objects in another SWF file. Cross-scripting is not supported between SWF files written using ActionScript 3.0 and files using previous versions of ActionScript; however, these files can communicate by using the LocalConnection class.

Memory usage and disk storage protections

Flash Player includes security protections for disk data and memory usage on the client computer.

The only type of persistent storage is through the SharedObject class, which is embodied as a file in a directory whose name is related to that of the owning SWF file. An application cannot typically write, modify, or delete any files on the client computer other than SharedObject data files, and it can only access SharedObject data files under the established settings per domain.

Flash Player helps limit potential denial-of-service attacks involving disk space (and system memory) through its monitoring of the usage of SharedObject classes. Disk space is conserved through limits automatically set by Flash Player (the default is 100K of disk space for each domain). The author can set the application to prompt the user for more disk space, or Flash Player automatically prompts the user if an attempt is made to store data that exceeds the limit. In either case, the disk space limit is enforced by Flash Player until the user gives explicit permission for an increased allotment for that domain.

Flash Player contains memory and processor safeguards that help prevent applications from taking control of excess system resources for an indefinite period of time. For example, Flash Player can detect an application that is in an infinite loop and select it for termination by prompting the user. The resources that the application uses are immediately released when the application closes.

Flash Player uses a garbage collector engine. The processing of new allocation requests always first ensures that memory is cleared so that the new usage always obtains only clean memory and cannot view any previous data.

Privacy

Privacy is an important aspect of overall security. Adobe products, including Flash Player, provide very little information that would reveal anything about a user (or their computer). Flash Player does not provide personal information about users (such as names, e-mail addresses, and phone numbers), or provide access to other sensitive information (such as credit card numbers or account information).

What Flash Player does provide is basically standardized hardware and software configuration information that authors might use to enhance the user experiences in the environment encountered. The same information is often available already from the operating system or web browser.

Information about the client environment that is available to the application includes:

- User agent string, which typically identifies the embedding browser type and operating system of the client
- System capabilities such as the language or the presence of an MP3 decoder (see the Capabilities class)
- Presence of a camera and microphone
- Keyboard and mouse input

ActionScript also includes the ability to replace the contents of the client's Clipboard by using the `setClipboard()` method of the System class. This method does not have a corresponding `getClipboard()` method, so protected data that might be stored in the Clipboard already is not accessible to Flash Player.

About sandboxes

The sandbox type indicates the type of security zone in which the SWF file is operating. In Flash Player, all SWF files (and HTML files, for the purposes of SWF-to-HTML scripting) are placed into one of four types of sandbox:

remote All files from non-local URLs are placed in a remote sandbox. There are many such sandboxes, one for each Internet (or intranet) domain from which files are loaded.

local-with-filesystem The default sandbox for local files. SWF files in this sandbox may not contact the Internet (or any servers) in any way—they may not access network endpoints with addresses such as HTTP URLs.

local-with-networking SWF file in this sandbox may communicate over the network but may not read from local file systems.

local-trusted This sandbox is not restricted. Any local file can be placed in this sandbox if given authorization by the end user. This authorization can come in two forms: interactively through the Settings Manager or noninteractively through an executable installer that creates Flash Player configuration files on the user's computer.

You can determine the current sandbox type by using the `sandboxType` property of the Security class, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- security/DetectCurrentSandbox.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script><![CDATA[
        [Bindable]
        private var mySandboxType:String;
        private function initApp():void {
            mySandboxType = String(Security.sandboxType);
        }
    ]]></fx:Script>
    <s:HGroup>
        <s:Label text="Sandbox Type: "/>
        <s:Label id="tb1" text="{mySandboxType}"/>
    </s:HGroup>
</s:Application>
```

When you compile an application, you have some control over which sandbox the application is in. This determination is a combination of the value of the `use-network` compiler option (the default is `true`) and whether the SWF file was loaded by the client over a network connection or as a local file.

The following table shows how the sandbox type is determined:

use-network	Loaded	Sandbox type
false	locally	local-with-filesystem
true	locally	local-with-network
true	network	remote
false	network	n/a (causes an error)

Browser security

Flash Player clients can be one of the following four types:

- Embedded Flash Player
- Debugger version of embedded Flash Player
- Stand-alone Flash Player
- Debugger version of stand-alone Flash Player

The stand-alone Flash Player runs on the desktop. It is typically used by people who are running applications that are installed and maintained by an IT department that has access to the desktop on which the application runs.

The embedded Flash Player is run within a browser. Anyone with Internet access can run applications from anywhere with this player. For Internet Explorer, the embedded player is loaded as an ActiveX control inside the browser. For Netscape-based browsers (including Firefox), it is loaded as a plug-in inside the browser. Using an embedded player lets the developer use browser-based technologies such as FORM and BASIC authentication as well as SSL.

Browser APIs

Applications hosting the Flash Player ActiveX control or Flash Player plug-in can use the EnforceLocalSecurity and DisableLocalSecurity API calls to control security settings. If DisableLocalSecurity is opened, the application does not benefit from the local-with-networking and local-with-file-system sandboxes. All files loaded from the local file system are placed into the local-trusted sandbox. The default behavior for an ActiveX control hosted in a client application is DisableLocalSecurity.

If EnforceLocalSecurity is opened, the application can use all three local sandboxes. The default behavior for the browser plug-in is EnforceLocalSecurity.

Cross-scripting

Cross-scripting is when a SWF file communicates directly with another SWF file. This communication includes calling methods and setting properties of the other SWF file.

SWF file loading and cross-scripting are always permitted between SWF files that reside in the same sandbox. For example, any local-with-filesystem SWF file can load and cross-script any other local-with-filesystem SWF file; any local-with-networking SWF file can load and cross-script any other local-with-networking SWF file; and so on. The restrictions appear when two SWF files from different sandboxes or two remote SWF files with different domains attempt to cooperate.

For SWF files in the remote sandbox, if two SWF files were loaded from the same domain, they can cross-script without any restrictions. If both SWF files were loaded from a network, but from different domains, you must provide permissions to allow them to cross-script.

To enable cross-scripting between SWF files, use the Security class's `allowDomain()` and `allowInsecureDomain()` methods.

You call these methods from the called SWF file and specify the calling SWF file's domain. For example, if SWF1 in domainA.com calls a method in SWF2 in domainB, SWF2 must call the `allowDomain()` method and specifically allow SWF files from domainA.com to cross-script the method, as the following example shows:

```
import flash.system.Security;
Security.allowDomain("domainA.com");
```

If the SWF files are in different sandboxes (for example, if one SWF file was loaded from the local file system and the other from a network) they must adhere to the following set of rules:

- Remote SWF files (those served over HTTP and other non-local protocols) can never load local SWF files.
- Local-with-networking SWF files can never load local-with-filesystem SWF files, or vice versa.
- Local-with-filesystem SWF files can never load remote SWF files.
- Local-trusted SWF files can load SWF files from any sandbox.

To facilitate SWF-to-SWF communication, you can also use the LocalConnection class. For more information, see "Using the LocalConnection class" on page 130.

ExternalInterface

You use the ExternalInterface API to let your application call scripts in the wrapper and to allow the wrapper to call functions in your application. The ExternalInterface API consists primarily of the `call()` and `addCallback()` methods in the flash.net package.

This communication relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` properties define. You set the values of the `allowScriptAccess` and `allowNetworking` properties in the SWF file's wrapper. For more information, see "About the object and embed tags" on page 2564.

By default, the application and the HTML page it is calling must be in the same domain for the `call()` method to succeed. For more information, see "Communicating with the wrapper" on page 226.

The navigateToURL() method

The navigateToURL() method opens or replaces a window in the Flash Player's container application. You typically use it to launch a new browser window, although you can also embed script in the method's call to perform other actions.

This usage of the `navigateToURL()` method relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` parameters define. You set the values of the `allowScriptAccess` and `allowNetworking` parameters in the SWF file's wrapper. For more information, see "About the object and embed tags" on page 2564.

Caching

Applications reside entirely on the client. If the browser loads the application, the application SWF file, plus externally loaded images and other media files, are stored locally on the client in the browser's cache. These files reside in the cache until cleared.

Storing a SWF file in the browser's cache can potentially expose the file to people who would not otherwise be able to see it. The following table shows some example locations of the browser's cache files:

Browser or operating system	**Cache location**
Internet Explorer on Windows XP	C:\Documents and Settings\username\Local Settings\Temporary Internet Files
Firefox on Windows XP	C:\Documents and Settings\username\Application Data\Mozilla\Firefox\Profiles\username.default\Cache
UNIX	$HOME/.mozilla/firefox/*username*.default/Cache/

These files can remain in the cache even after the browser is closed.

To prevent client browsers from caching the SWF file, try setting the following HTTP headers in the SWF file's wrapper:

```
Cache-control: no-cache, no-store, must-revalidate, max-age=-1
Pragma: no-cache, no-store
Expires: -1
```

Note that in some cases, setting the `Pragma` header to `"no-cache"` can cause a runtime error with GET requests over SSL with the HTTPService class. In this case, setting just the `Cache-control` header should work.

Marc Speck has a blog entry that describes the issue and presents several solutions.

Trusted sites and directories

The browser security model includes levels of trust applied to specific websites. Flash Player interacts with this model by assigning a sandbox based on whether the browser declared the site of the SWF file's origin trusted.

If Flash Player loads a SWF file from a trusted website, the SWF file is put in the local-trusted sandbox. The SWF file can read from local data sources and communicate with the Internet.

You can also assign a SWF file to be in the local-trusted sandbox when you load it from the local file system. To do this, you configure a directory as trusted by Flash Player (which results in the SWF file being put in the local-trusted sandbox) by adding a FlashPlayerTrust configuration file that specifies the directory to trust. This requires administrator access privileges to the client system, so it is typically used in controlled environments. Users can also define a directory as trusted by using the Flash Player User Settings Manager. For more information, see the Flash Player documentation.

Deploying secure applications

When you deploy an application, you make the application accessible to your users. The process of deploying an application is dependent on your application, your application requirements, and your deployment environment. You can employ some of the following strategies to ensure that the application you deploy is secure.

Deploying local SWF files versus network SWF files

Client computers can obtain individual SWF files from a number of sources, such as from an external website or a local file system. When SWF files are loaded into Flash Player, they are individually assigned to security sandboxes based on their origin.

Flash Player classifies SWF files downloaded from the network (such as from external websites) in separate sandboxes that correspond to their website origin domains. By default, these files are authorized to access additional network resources that come from the specific (exact domain name match) site. Network SWF files can be allowed to access additional data from other domains by explicit website and author permissions.

A local SWF file describes any file referenced by using the "file" protocol or a UNC path, which does not include an IP address or a qualifying domain. For example, "\\test\test.swf" and "file:\test.swf" are considered local files, while "\\test.com\test.swf" and "\192.168.0.1\test.swf" are not considered local files.

Local SWF files from local origins, such as local file systems or UNC network paths, are placed into one of three sandboxes: local-with-networking, local-with-filesystem, and local-trusted.

When you compile the application, if you set the `use-network` compiler option to `false`, local SWF files are placed in the local-with-filesystem sandbox. If you set the `use-network` compiler option to `true`, local SWF files are placed in the local-with-networking sandbox.

Local SWF files that are registered as trusted (by users or by installer programs) are placed in the local-trusted sandbox. Users can also reassign (move) a local SWF file to or from the local-trusted sandbox based on their security considerations.

Deploy checklist

Before you deploy your application, ensure that your proxy servers, firewalls, and assets are configured properly. Adobe provides a deployment checklist that you can follow. For more information, see "Deployment checklist" on page 2548.

Remove wildcards

If your application relies on assets loaded from another domain, and that domain has a crossdomain.xml file on it, remove wildcards from that file if possible. For example, change the following:

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all"/>
    <allow-access-from domain="*" to-ports="*"/>
</cross-domain-policy>
```

to this:

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="by-content-type"/>
    <allow-access-from domain="*.myserver.com" to-ports="80,443,8100,8080" />
</cross-domain-policy>
```

Also, set the value of the `to-ports` attribute of the `allow-access-from` tag to ensure that you are only allowing necessary ports access to the resources.

Check your application for calls to the `allowDomain()` and `allowInsecureDomain()` methods. During development, you might pass these methods a wildcard character (*), but now restrict those methods to allowing requests only from the necessary domains.

Deploy assets to WEB-INF

In some deployments, you want to make assets such as data files accessible to the application, but not accessible to anyone requesting the file. If you are using a J2EE-based server, you can deploy those files to a subdirectory within the WEB-INF directory. Based on J2EE security constraints, no J2EE server can return a resource from the WEB-INF directory to any client request. The only way to access files in this directory is with server-side code.

Loading assets

The most common task that developers perform that requires an understanding of security is loading external assets.

Data compared to content

The Flash Player security model makes a distinction between loading content and accessing or loading data. Content is defined as media: visual media that Flash Player can display, such as audio, video, or a SWF file that includes displayed media. Data is defined as something that you can manipulate only with ActionScript code.

You can load data in one of two ways: by extracting data from loaded media content, or by directly loading data from an external file (such as an XML file) or socket connection. You can extract data from loaded media by using the `BitmapData.draw()` method, the `Sound.id3` property, or the `SoundMixer.computeSpectrum()` method. You can load data by using classes such as the SWFLoader, URLStream, URLLoader, Socket, and XMLSocket classes.

The Flash Player security model defines different rules for loading content and accessing data. Loading content has fewer restrictions than accessing data. In general, content such as SWF files, bitmaps, MP3 files, and videos can be loaded from anywhere, but if the content is from a domain other than that of the loading SWF file, it will be partitioned in a separate security sandbox.

When you load sub applications into a main application with the SWFLoader control, the sandbox into which you load it determines the level of interoperability between the applications. For more information, see “About security domains” on page 179.

Loading remote assets

Loading remote or network assets relies on three factors:

- Type of asset. If the target asset is a content asset, such as an image file, you do not need any specific permissions from the target domain to load its assets into your application. If the target asset is a data asset, such as an XML file, you must have the target domain's permission to access this asset. For more information on the types of assets, see "Data compared to content" on page 124.
- Target domain. If you are loading data assets from a different web domain, the target domain must provide a crossdomain.xml policy file. This file contains a list of URLs and URL patterns that it allows access from. The calling domain must match one of the URLs or URL patterns in that list. If the target asset is a SWF file, you can also provide permissions by calling the `loadPolicyFile()` method and loading an alternative policy file inside that target SWF file. For more information, see "Using cross-domain policy files" on page 125.
- Loading SWF file's sandbox. To load an asset from a network address, you must ensure that your SWF file is in either the remote or local-with-networking sandbox. To ensure that a SWF file can load assets over the network, you must set the `use-network` compiler option to `true` when you compile the application. This is the default. If the application was loaded from the local file system with `use-network` set to `false`, the application is put in the local-with-filesystem sandbox and it cannot load remote SWF files.

Loading assets from a remote location that you do not control can potentially expose your users to risks. For example, the remote website B contains a SWF file that is loaded by your website A. This SWF file normally displays an advertisement. However, if website B is compromised and its SWF file is replaced with one that asks for a username and password, some users might disclose their login information. To prevent data submission, the loader has a property called `allowNetworking` with a default value of `never`.

Using cross-domain policy files

To make data available to SWF files in different domains, use a *cross-domain policy file*. A cross-domain policy file is an XML file that provides a way for the server to indicate that its data and documents are available to SWF files served from other domains. Any SWF file that is served from a domain that the server's policy file specifies is permitted to access data or assets from that server.

When a Flash document attempts to access data from another domain, Flash Player attempts to load a policy file from that domain. If the domain of the Flash document that is attempting to access the data is included in the policy file, the data is automatically accessible.

The default policy file is named crossdomain.xml and resides at the root directory of the server that is serving the data. The following example policy file permits access to Flash documents that originate from foo.com, friendOfFoo.com, *.foo.com, and 105.216.0.40:

```
<?xml version="1.0"?>
<!-- http://www.foo.com/crossdomain.xml -->
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="by-content-type"/>
    <allow-access-from domain="www.friendOfFoo.com"/>
    <allow-access-from domain="*.foo.com"/>
    <allow-access-from domain="105.216.0.40"/>
</cross-domain-policy>
```

You can also configure ports in the crossdomain.xml file. For more information about crossdomain.xml policy files, see *Security*.

You can use the `loadPolicyFile()` method to access a nondefault policy file.

Loading local assets

In some cases, your SWF file might load assets that reside on the client's local file system. This typically happens when the application is embedded on the client device and loaded from a network. If the application is allowed to access local assets, it cannot access network assets.

To ensure that an application can access assets in the local sandbox, the application must be in the local-with-filesystem or local-trusted sandbox. To ensure this, you set the `use-network` compiler option to `false` when you compile the application. The default value of this option is `true`.

When you load another SWF file that is in the local file system into your application with a class such as SWFLoader, and you want to call methods or access properties of that SWF file, you do not need to explicitly enable cross-scripting.

If the SWF files are in different sandboxes (for example, you loaded the main SWF file into the local-with-network sandbox, but loaded the asset SWF file from the network), you cannot cross-script because they are in different sandboxes. Remote SWF files cannot load local SWF files, and vice versa.

Using J2EE authentication

Applications built with Flex integrates well with any server environment, including J2EE. To effectively implement secure web applications in a J2EE environment, you should understand the following concepts:

Authentication The process of gathering user credentials (user name and password) and validating them in the system. This requires checking the credentials against a user repository such as a database, flat file, or LDAP implementation, and authenticating that the user is who they say they are.

Authorization The process of making sure that the authenticated user is allowed to view or access a given resource. If a user is not authorized to view a resource, the container does not allow access.

Using container-based authentication

J2EE uses the Java Authentication and Authorization Service (JAAS), Java security manager, and policy files to enforce access controls on users and ties this enforcement to web server roles. The authenticating mechanism is role based. That is, all users who access a web application are assigned to one or more roles. Example roles are manager, developer, and customer.

Application developers can assign usage roles to a web application, or to individual resources that make up the application. Before a user is granted access to a web application resource, the container ensures that the user is identified (logged in) and that the user is assigned to a role that has access to the resource. Any unauthorized access of a web application results in an HTTP 401 (Unauthorized) status code.

Authentication requires a website to store information about users. This information includes the role or roles assigned to each user. In addition, websites that authenticate user access typically implement a login mechanism that forces verification of each user's identity by using a password. After the website validates the user, the website can then determine the user's roles.

This logic is typically implemented in one of the following forms:

- JDBC Login Module
- LDAP Login Module
- Windows Login Module
- Custom JAAS Login Module

Authentication occurs on a per-request basis. The container typically checks every request to a web application and authenticates it.

Authentication requires that the roles that the application developer defines for a web application be enforced by the server that hosts the application.

As part of developing and deploying an application, you must configure the following application authentication settings:

- Access roles to applications
- Resource protection
- Application server validation method

The web application's deployment descriptor, web.xml, contains the settings for controlling application authentication. This file is stored in the web application's WEB-INF directory.

Using authentication to control access to applications

To use authentication to prevent unauthorized access to your application, you typically use the container to set up constraints on resources. You then challenge the user who then submits credentials. These credentials determine the success or failure of the user's login attempt, as the container's authentication logic determines.

For example, you can protect the page that the application is returned with, or protect the SWF file itself. You do this in the web.xml file by defining specific URL patterns, as the following example shows:

```
<web-app>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Payroll Application</web-resource-name>
                <url-pattern>/payroll/*</url-pattern>
                <http-method>GET</http-method>
                <http-method>POST</http-method>
            </web-resource-collection>
        <auth-constraint>
            <role-name>manager</role-name>
        </auth-constraint>
    </security-constraint>
</web-app>
```

When the browser tries to load a resource that is secured by constraints in the web.xml file, the browser either challenges the user (if you are using BASIC authentication) or forwards the user to a login page (with FORM authentication).

With BASIC authentication, the user enters a username and password in a popup box that the browser creates. To specify that an application uses BASIC authentication, you use the `login-config` element and its `auth-method` subelement in the web application's web.xml file, as the following example shows:

```
<web-app>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>Managers</realm-name>
    </login-config>
    ...
</web-app>
```

With FORM authentication, you must code the page that accepts the username and password, and submit them as FORM variables named `j_username` and `j_password`. This form can be implemented in HTML or as an application or anything that can submit a form.

When you configure FORM authentication, you can specify both a login form and an error form in the web.xml file, as the following example shows:

```
<web-app>
    <login-config>
        <auth-method>FORM</auth-method>
        <form-login-config>
            <form-login-page>/login.htm</form-login-page>
            <form-error-page>/loginerror.htm</form-error-page>
        </form-login-config>
    </login-config>
</web-app>
```

You submit the results of the form validation to the `j_security_check` action. The server executing the application recognizes this action and processes the form.

A simple HTML-based form might appears as follows:

```
<form method="POST" action="j_security_check">
    <table>
        <tr><td>User</td><td><input type=text name="j_username"></tr>
        <tr><td>Password</td><td><input type=password name="j_password"></tr>
    </table>
    <input type=submit>
</form>
```

The results are submitted to the container's JAAS system with base-64 encoding, which means they can be read by anyone that can view the TCP/IP traffic. Use encryption to prevent these so-called "man-in-the-middle" attacks. In both BASIC and FORM authentication, if the user accessed the resource through SSL, the username and password submission are encrypted, as is all traffic during that exchange.

After it is complete, the container populates the browser's security context and provides or denies access to the resource. Flash Player inherits the security context of the underlying browser. As a result, when you make a data service call, the established credentials are used.

When a user fails an authentication attempt with invalid credentials, be sure not to return information about which item was incorrect. Instead, use a generic message such as "Your login information was invalid."

Using RPC services

You can use the RPC services classes—RemoteObject, HTTPService, and WebService—not only to control access to the data that goes into an application, but also to control the data and actions that flow out of it. You can also use service authentication to allow only certain users to perform certain actions. For example, if you have an application that allows employee data to be modified through a RemoteObject call, use RemoteObject authentication to make sure that only managers can change the employee data.

A service-based architecture makes it easy to implement several different security models for your application. You can use programmatic security to limit access to services, or you can apply declarative security constraints to entire services.

When accessing RPC services with Flex tags such as the `<mx:WebService>` and `<mx:HTTPService>` tags, your application's SWF file must connect to the service directly, which means that it can encounter security-based limitations. When using RPC services, one of the following must be true:

- The RPC is in the same domain as the application that calls it.
- The RPC's host system has a crossdomain.xml file that explicitly allows access from the application's domain.

Using secured services

Secured services are services that are protected by resource constraints. The service itself behaves as a resource that needs authentication and the container defines its URL pattern as requiring authorization.

You might have a protected application that calls a protected resource. In this case, with BASIC authentication and a proxied destination, the user's credentials are passed through to the service. The user only has to log on once when they first start the application, and not when the application attempts to access the service.

Without a proxy, the user is challenged to enter their credentials a second time when the application attempts to access the service.

When you use secured services, keep the following in mind:

- If possible, use HTTPS for your services when you use authentication. In BASIC and custom authentication, user names and passwords are sent in a base-64 encoding. Using base-64 encoding hides the data only from plain view; HTTPS actually encrypts the data. You can use HTTPS in these cases by making sure HTTPS is set up on your server and by adding a protocol attribute with the value `https` on the service, and by adding a crossdomain.xml file.
- To ensure that the WebService and HTTPService endpoints are secure, use a browser window to access the URL you are trying to secure. This should always bring up a BASIC authentication prompt.
- If the BASIC or custom login box appears but you can't log in, make sure that the users and roles were added correctly to your application server. This is often an error-prone task that is overlooked as the source of the problem.

Making other connections

Flash Player can connect to servers, services, and load data from sources other than RPC services. Some of these sources have security issues that you should consider.

Using RTMP

Flash Player uses the Real-Time Messaging Protocol (RTMP) for client-server communication. This is a TCP/IP protocol designed for high-performance transmission of audio, video, and data messages. RTMP sends unencrypted data, including authentication information (such as a name and a password).

Although RTMP in and of itself does not offer security features, Flash communications applications can perform secure transactions and secure authentication through an SSL-enabled web server.

RTMPT connections are HTTP connections for the client to the server over which RTMP data is tunneled. When a direct RTMP connection is unavailable, the standard and secure channels use RTMPT and tunneled RTMPS connections, respectively, on the RTMP endpoint.

Use the secure RTMP channel to connect to the RTMP endpoint over Transport Layer Security (TLS). This channel supports real-time messaging and server-pushed broadcasts. This channel falls back to tunneled RTMPS when a direct connection is unavailable.

Using sockets

Sockets let you read and write raw binary or XML data with a connected server. Sockets transmit over TCP. Because of this, Flash Player cannot take advantage of the built-in encryption capabilities of the browser. However, you can use encryption algorithms written in ActionScript to protect the data that is being communicated.

Cross-domain access to socket and XML socket connections is disabled by default. Access to socket connections in the same domain of the SWF file on ports lower than 1024 is also disabled by default. You can permit access to these connections by serving a cross-domain policy file from any of the following locations:

- The same port as the main socket connection
- A different port
- The HTTP server on port 80 in the same domain as the socket server

For more information, see the Socket and XMLSocket classes in ActionScript 3.0 Reference for the Adobe Flash Platform.

Using the LocalConnection class

The LocalConnection class lets you develop SWF files that can send instructions to each other. LocalConnection objects can communicate only among SWF files that are running on the same client computer, but they can be running in different applications—for example, a SWF file running in a browser and a SWF file running in a projector. (A projector is a SWF file saved in a format that can run as a stand-alone application—that is, the projector doesn't require Flash Player to be installed since it is embedded inside the executable file.)

For every LocalConnection communication, there is a sender SWF file and a listener SWF file. The simplest way to use a LocalConnection object is to allow communication only between LocalConnection objects located in the same domain because you won't have security issues.

Applications served from different domains that need to be able to make LocalConnection calls to each other must be granted cross-domain LocalConnection permissions. To do this, the listener must allow the sender permission by using the `LocalConnection.allowDomain()` or `LocalConnection.allowInsecureDomain()` methods.

Adobe does not recommend using the `LocalConnection.allowInsecureDomain()` method because allowing non-HTTPS documents to access HTTPS documents compromises the security offered by HTTPS. It is best that all Flash SWF files that make LocalConnection calls to HTTPS SWF files are served over HTTPS.

For more information about using the LocalConnection class, see *ActionScript 3.0 Developer's Guide*.

To facilitate SWF-to-SWF communication, you can also use cross-scripting. For more information, see "Cross-scripting" on page 121.

Using SSL

A SWF file playing in a browser has many of the same security concerns as an HTML page being displayed in a browser. This includes the security of the SWF file while it is being loaded into the browser, as well as the security of communication between Flash and the server after the SWF file has loaded and is playing in the browser. In particular, data communication between the browser and the server is susceptible to being intercepted by third parties. The solution to this issue in HTML is to encrypt the communication between the client and server to make any data captured by third parties undecipherable and thus unusable. This encryption is done by using an SSL-enabled browser and server.

Because a SWF file running within a browser uses the browser for almost all of its communication with the server, it can take advantage of the browser's built-in SSL support. This lets communication between the SWF file and the server be encrypted. Furthermore, the actual bytes of the SWF file are encrypted while they are being loaded into the browser. Thus, by playing a SWF file within an SSL-enabled browser through an HTTPS connection with the server, you can ensure that the communication between Flash Player and the server is encrypted and secure.

The one exception to this security is the way Flash Player uses persistent sockets (through the ActionScript XMLSocket object), which does not use the browser to communicate with the server. Because of this, SWF files that use sockets cannot take advantage of the built-in encryption capabilities of the browser. However, you can use one-way encryption algorithms written in ActionScript to encrypt the data being communicated.

MD5 is a one-way encryption algorithm described in RFC 1321. This algorithm has been ported to ActionScript, which enables developers to secure one-way data by using the MD5 algorithm before it is sent from the SWF file to the server. For more information about RFC 1321, see www.faqs.org/rfcs/rfc1321.html.

Using secure endpoints

To access HTTP services or web services through HTTPS, you can specify the protocols using “https” in the `wsdl` or `url` properties; for example:

```
<mx:WebService url="https://myservice.com" .../>
<mx:HTTPService wsdl="https://myservice.com" .../>
```

By default, a SWF file served over an unsecure protocol, such as HTTP, cannot access other documents served over the secure HTTPS protocol, even when those documents come from the same domain. As a result, if you loaded the SWF file over HTTP but want to connect to the service through HTTPS, you must add `secure="false"` in the crossdomain.xml file on the services’s server, as the following example shows:

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="by-content-type"/>
    <allow-access-from domain="*.mydomain.com" secure="false"/>
</cross-domain-policy>
```

If you loaded the SWF file over HTTPS, you do not have to make any changes.

Writing secure applications

When you code an application, keep the following topics in mind to ensure that the application you write is as secure as possible.

MXML tags with security restrictions

Some MXML tags trigger operations that require security settings. Operations that trigger security checks include:

- Referencing a URL that is outside the exact domain of the application that makes a request.
- Referencing an HTTPS URL when the application that makes the request is not served over HTTPS.
- Referencing a resource that is in a different sandbox.

In these cases, access rights must be granted through one of the permission-granting mechanisms such as the `allowDomain()` method or a crossdomain.xml file.

MXML tags that can trigger security checks include:

- Any class that extends the Channel class.
- RPC-related tags that use channels such as `<mx:WebService>`, `<mx:RemoteObject>`, and `<mx:HTTPService>`.
- Messaging tags such as `<mx:Producer>` and `<mx:Consumer>`.
- The `<mx:DataService>` tag.
- Tags that load SWF files such as `<mx:SWFLoader>` and `<s:ModuleLoader>`.

In addition to these tags and their underlying classes, many Flash classes trigger security checks including ExternalInterface, Loader, NetStream, SoundMixer, URLLoader, and URLRequest.

Disabling viewSourceURL

If you enabled the view source feature by setting the value of the `viewSourceURL` property on the `<s:Application>` tag, you must be sure to remove it before you put your application into production.

This functionality applies only to Flash Builder users.

Remove sensitive information from SWF files

Applications built with Flash share many of the same concerns and issues as web pages when it comes to protecting the security of data. Because the SWF file format is an open format, you can extract data and algorithms contained within a SWF file. This is similar to how HTML and JavaScript code can be easily viewed by users. However, SWF files make viewing the code more difficult. A SWF file is compiled and is not human-readable like HTML or JavaScript.

But security is not obtained through obscurity. A number of third-party tools can extract data from compiled SWF files. As a result, do not consider that any data, variables, or ActionScript code compiled into an application are secure. You can use a number of techniques to secure sensitive information and still make it available for use in your SWF files.

To help ensure a secure environment, use the following general guidelines:

- Do not include sensitive information, such as user names, passwords, or SQL statements in SWF files.
- Do not use client-side username and password checks for authentication.
- Remove debug code, unused code, and comments from code before compiling to minimize the amount of information about your application that is available to someone with a decompiler or a debugger version of Flash Player.
- If your SWF file needs access to sensitive information, load the information into the SWF file from the server at run time. The data will not be part of the compiled SWF file and thus cannot be extracted by decompiling the SWF file. Use a secure transfer mechanism, such as SSL, when you load the data.
- Implement sensitive algorithms on the server instead of in ActionScript.
- Use SSL whenever possible.
- Only deploy your web applications from a trusted server. Otherwise, the server-side aspect of your application could be compromised.

Input validation

Input validation means ensuring that input is what it says it is or is what it is supposed to be. If your application is expecting name and address information, but it gets SQL commands, have a validation mechanism in your application that checks for and filters out SQL-specific characters and strings before passing the data to the execute method.

In many cases, you want users to provide input in TextInput, TextArea, and other controls that accept user input. If you use the input from these controls in operations inside the application, make sure that the input is free of possible malicious characters or code.

One approach to enforcing input validation is to use the validator classes. Validators ensure that the input conforms to a predetermined pattern. For example, the NumberValidator class ensures that a string represents a valid number. This validator can ensure that the input falls within a given range (specified by the `minValue` and `maxValue` properties), is an integer (specified by the `domain` property), is non-negative (specified by the `allowNegative` property), and does not exceed the specified precision.

In typical client-server environments, data validation occurs on the server after data is submitted to it from the client. One advantage of using validators in Flex is that they execute on the client, which lets you validate input data before transmitting it to the server. By using validators, you eliminate the need to transmit data to and receive error messages back from the server, which improves the overall responsiveness of your application.

You can also write your own ActionScript filters that remove potentially harmful code from input. Common approaches include stripping out dollar sign ($), quotation mark ("), semi-colon (;) and apostrophe (') characters because they have special meaning in most programming languages. Because Flex also renders HTML in some controls, also filter out characters that can be used to inject script into HTML, such as the left and right angle brackets ("<" and ">"), by converting these characters to their HTML entities "&lt;" and "&gt;". Also filter out the left and right parentheses ("("and ")") by translating them to "&#40;" and "&#41;", and the pound sign ("#") and ampersand ("&") by translating them to "&#35" (#) and "&#38" (&).

Another approach to enforcing input validation is to use strongly-typed, parameterized queries in your SQL code. This way, if someone tries to inject malicious SQL code into text that is used in a query, the SQL server will reject the query.

For more information on potentially harmful characters and conversion processes, see http://www.cert.org/tech_tips/malicious_code_mitigation.html.

For more information about validators, see "Validating Data" on page 1964.

ActionScript

Use some of the following techniques to try to make your use of ActionScript more secure.

Handling errors

The SecurityError exception is thrown when some type of security violation takes place. Security errors include:

- An unauthorized property access or method call was made across a security sandbox boundary.
- An attempt was made to access a URL not permitted by the security sandbox.
- A socket connection was attempted to an unauthorized port number, for example, a port below 1024, without a policy file present.
- An attempt was made to access the user's camera or microphone, and the request to access the device was denied by the user.

Flash Player dispatches SecurityErrorEvent objects to report the occurrence of a security error. Security error events are the final events dispatched for any target object. This means that any other events, including generic error events, are not dispatched for a target object that experiences a security error.

Your event listener can access the SecurityErrorEvent object's `text` property to determine what operation was attempted and any URLs that were involved, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- security/SecurityErrorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import flash.net.URLLoader;
        import flash.net.URLRequest;
        import flash.events.SecurityErrorEvent;
        import mx.controls.Alert;
        private var loader:URLLoader = new URLLoader();

        private function triggerSecurityError():void {
            var request:URLRequest = new URLRequest("c:/temp/feeds.txt");

            // Triggers a security exception if you run this SWF from a server without
            // explicitly allowing local file-access permissions.
           try {
                loader.load(request);
           } catch (error:SecurityError) {
                Alert.show(error.name + ": " + error.message);
           }
        }
    ]]></fx:Script>
    <s:Button id="b1" label="Click Me To Trigger Security Error"
click="triggerSecurityError()"/>
</s:Application>
```

If no event listeners are present, the debugger version of Flash Player automatically displays an error message that contains the contents of the text property.

In general, try to wrap methods that might trigger a security error in a try/catch block. This prevents users from seeing information about destinations or other properties that you might not want to be visible.

Suppressing debug output

Flash Player writes debug output from a `trace()` method or the Logging API to a log file on the client. Any client can be running the debugger version of Flash Player. As a result, remove calls to the `trace()` method and Logging API calls that produce debugging output so that clients cannot view your logged information.

If you use the Logging API in your custom components and classes, set the value of the LogEventLevel to `NONE` before compilation, as the following example shows:

```
myTraceTarget.level = LogEventLevel.NONE;
```

For more information about the Logging API, see “Using the logging API” on page 2229.

Using host-based authentication

IP addresses and HTTP headers are sometimes used to perform host-based authentication. For example, you might check the `Referer` header or the client IP address to ensure that a request comes from a trusted source.

However, request headers such as Referer can be spoofed easily. This means that clients can pretend to be something they are not by settings headers or faking IP addresses. The solution to the problem of client spoofing is to not use HTTP header data as an authentication mechanism.

Using passwords

Using passwords in your application is a common way to protect resources from unauthorized access. Test the validity of the password on the server rather than the client, because the client has access to all the logic in the local SWF file.

Never store passwords locally. For example, do not store username and password combinations in local SharedObjects. These are stored in plain-text and unencrypted, just as cookie files are. Anyone with access to the user's computer can access the information inside a SharedObject.

To ensure that passwords are transmitted from the client to the server safely, enforce the use of SSL or some other secure transport-level protocol.

When you ask for a password in a TextArea or TextInput control, set the displayAsPassword property to true. This displays the password as asterisks as it is typed. This also prevents copy/paste operations from accessing the underlying characters in the text field.

Storing persistent data with the SharedObject class

Flash Player supports persistent shared objects through the SharedObject class. The SharedObject class stores data on users' computers. This data is usually local, meaning that it was obtained with the SharedObject.getLocal() method. You can also create persistent remote data with the SharedObject class; this requires Adobe® Flash® Media Server.

Each remote sandbox has an associated store of persistent SharedObject directory on the client. For example, when any SWF from domain1.com reads or writes data with the SharedObject class, Flash Player reads or writes that object in the domain1.com object store. Likewise for a SWF from domain2.com, Flash Player uses the domain2.com store. To avoid name collisions, the directory path defaults to the full path in the URL of the creating SWF file. This process can be shortened by using the localPath parameter of the SharedObject.getLocal() method, which allows other SWF files from the same domain to access a shared object after it is created.

Every domain has a maximum amount of data that a SharedObject class can save in the object store. This is an allocation of the user's disk space in which applications from that domain can store persistent data. Users can change the quota for a domain at any time by choosing Settings from the Flash Player context menu. When an application tries to store data with a SharedObject class that causes Flash Player to exceed its domain's quota, a dialog box appears, asking the user whether to increase the domain quota.

Configuring client security settings

Some security control features in Flash Player target user choices, and some target the modern corporate and enterprise environments, such as when the IT department would like to install Flash Player across the enterprise but has concerns about IT security and privacy. To help address these types of requirements, Flash Player provides various installation-time configuration choices. For example, some corporations do not want Flash Player to have access to the computer's audio and video hardware; other environments do not want Flash Player to have any read or write access to the local file system.

Three groups can make security choices: the application author (using developer controls), the administrative user (using administrator controls), and the local user (with user controls).

About the mm.cfg file

You configure the debugger version of Flash Player by using the settings in the mm.cfg text file. You must create this file when you first configure the debugger version of Flash Player.

The settings in this file let you enable or disable `trace()` logging, set the location of the `trace()` file's output, and configure client-side error and warning logging.

For more information, see "Editing the mm.cfg file" on page 2224.

About the mms.cfg file

The primary purpose for the security configuration file (mms.cfg) is to support the corporate and enterprise environments where the IT department wants to install Flash Player across the enterprise, while enforcing some common global security and privacy settings (supported with installation-time configuration choices).

On operating systems that support the concept of user security levels, the file is flagged as requiring system administrator (or root) permissions to modify or delete it. The following table shows the location of the mms.cfg file, depending on the operating system:

Operating System	**Location of mms.cfg file**
Macintosh OS X	/Library/Application Support/Macromedia
Windows XP/Vista	C:\WINDOWS\system32\Macromed\Flash
Windows 2000	C:\WINNT\System32\Macromed\Flash
Windows 95/98/ME	C:\WINDOWS\System\Macromed\Flash
Linux	/etc/adobe

You can use this file to configure security settings that deal with data loading, privacy, and local file access. The settings include:

- `FileDownloadDisable`
- `FileUploadDisable`
- `LocalStorageLimit`
- `AVHardwareDisable`

For a complete list of options and their descriptions, see http://www.adobe.com/go/fp9_admin.

About FlashPlayerTrust files

Flash Player provides a way for administrative users to register certain local files so that they are always loaded into the local-trusted sandbox. Often an installer for a native application or an application that includes many SWF files will do this. Depending on whether Flash Player will be embedded in a nonbrowser application, one of two strategies can be appropriate: register SWF files and HTML files to be trusted, or register applications to be trusted. Only applications that embed the browser plug-ins can be trusted—the stand-alone players and standard browsers do not check to see if they were trusted.

The installer creates files in a directory called FlashPlayerTrust. These files list paths of trusted files. This directory, known as the Global Flash Player Trust directory, is alongside the mms.cfg file, in the following location, which requires administrator access:

- Windows: system\Macromed\Flash\FlashPlayerTrust (for example, C:\winnt\system32\Macromed\Flash\FlashPlayerTrust)

- OS X: app support/Macromedia/FlashPlayerTrust (for example, /Library/Application Support/Macromedia/FlashPlayerTrust)

These settings affect all users of the computer. If an installer is installing an application for all users, the installer can register its SWF files as trusted for all users.

For more information about FlashPlayerTrust files, see http://www.adobe.com/devnet/flashplayer/articles/flash_player10_security_wp.html.

About the Settings Manager

The Settings Manager allows the individual user to specify various security, privacy, and resource usage settings for applications executing on their client computer. For example, the user can control application access to select facilities (such as their camera and microphone), or control the amount of disk space allotted to a SWF file's domain. The settings it manages are persistent and controlled by the user.

The user can indicate their personal choices for their Flash Player settings in a number of areas, either globally (for Flash Player itself and all applications built with Flash) or specifically (applying to specific domains only). To designate choices, the user can select from the six tab categories along the top of the Settings Manager dialog box:

- Global Privacy Settings
- Global Storage Settings
- Global Security Settings
- Flash Player Update Settings
- Privacy Settings for Individual Websites
- Storage Settings for Individual Websites

Access the Settings Manager for your Flash Player

1. Open an application in Flash Player.
2. Right-click and select Settings.

   The Adobe Flash Player Settings dialog box appears.
3. Select the Privacy tab (on the far left).
4. Click the Advanced button.

   Flash Player launches a new browser window and loads the Settings Manager help page.

Other resources

The following table lists resources that are useful in understanding the Flash Player security model and implementing security in your applications:

Resource name	Location
Security Topic Center	http://www.adobe.com/devnet/security
Security Bulletins and Advisories	http://www.adobe.com/support/security
Flash Player Security & Privacy	http://www.adobe.com/products/flashplayer/security
Security Resource Center	http://www.adobe.com/resources/security
Flash Player 10 Security white paper	http://www.adobe.com/devnet/flashplayer/articles/flash_player10_security_wp.html

Resource name	Location
Flash Player 10 Security changes	http://www.adobe.com/go/fp10_security
"Flash Player Security"	http://www.adobe.com/go/progAS3_security
"Networking and Communications"	http://www.adobe.com/go/AS3_networking_and_communications
Settings Manager	http://www.adobe.com/support/flashplayer/help/settings/

Modular applications

Modules quick start

Modules are SWF files that are similar to applications. You use modules to externalize functionality and load it only when it is needed. Applications can load and unload modules at run time, which can save start up time and memory because the applications can be smaller and download faster.

The most common use of modules is in a navigator container. Typically, each view in a navigator container is a module. When the user navigates to a new view, a new module is loaded.

You compile modules just as you would any application file. On the command line, you create a new module and compile it with the mxmlc compiler. In Flash Builder, you create a new module by selecting File > New > MXML Module.

Modules are MXML documents with `<s:Module>` as the root tag. Within that tag, you can use any child tags that you can use in a `<s:Application>` tag. The following examples shows two modules:

HorizontalLayoutModule.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- modules/mxmlmodules/HorizontalLayoutModule.mxml -->
<s:Module xmlns:fx="http://ns.adobe.com/mxml/2009"
          xmlns:s="library://ns.adobe.com/flex/spark"
          xmlns:mx="library://ns.adobe.com/flex/mx"
          minWidth="955" minHeight="600"
          >
    <fx:Declarations>
    </fx:Declarations>
    <fx:Style>
    </fx:Style>

    <fx:Script>
        <![CDATA[
        ]]>
    </fx:Script>
    <s:layout>
        <s:HorizontalLayout />
    </s:layout>
    <s:Label text="label three"/>
    <s:Button label="button three"/>
    <s:Label text="label four"/>
    <s:Button label="button four"/>
</s:Module>
```

VerticalLayoutModule.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- modules/mxmlmodules/VerticalLayoutModule.mxml -->
<s:Module xmlns:fx="http://ns.adobe.com/mxml/2009"
          xmlns:s="library://ns.adobe.com/flex/spark"
          xmlns:mx="library://ns.adobe.com/flex/mx"
          minWidth="955" minHeight="600">
    <fx:Declarations>
    </fx:Declarations>
    <fx:Style>
    </fx:Style>

    <fx:Script>
        <![CDATA[
        ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout />
    </s:layout>
    <s:Label text="label one"/>
    <s:TextInput />
    <s:Label text="label two"/>
    <s:TextArea/>
</s:Module>
```

To load modules in an application, you use a `<s:ModuleLoader>` tag inside a container. The container can be an MX container (such as a TabNavigator container), or a Spark container (such as a Group). The following example application uses a TabNavigator container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- modules/SimpleLoader.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600">

    <mx:TabNavigator width="500" height="300">
        <s:ModuleLoader label="Tab One" url="mxmlmodules/VerticalLayoutModule.swf"/>
        <s:ModuleLoader label="Tab Two" url="mxmlmodules/HorizontalLayoutModule.swf"/>
    </mx:TabNavigator>
</s:Application>
```

When the application first loads, Flash Player loads the first module in the first view of the container. When the user navigates to the second view of the container, Flash Player loads the second module.

The Spark ModuleLoader class implements the INavigatorContent interface. As a result, it provides some convenience properties for working with navigator containers. For example, you can use the `label` property to set a label on the tab in a TabNavigator container, or you can use the `icon` property to instead add an icon to the tab.

The ModuleLoader class also provides a `creationPolicy` property. You can use this property to instruct the application when to load the module.

More Help topics

"Using mxmlc, the application compiler" on page 2174

"About the creationPolicy property" on page 2336

Modular applications overview

About modules

Modules are SWF files that can be loaded and unloaded by an application. They cannot be run independently of an application, but any number of applications can share the modules.

Modules let you split your application into several pieces, or modules. The main application, or shell, can dynamically load other modules that it requires, when it needs them. It does not have to load all modules when it starts, nor does it have to load any modules if the user does not interact with them. When the application no longer needs a module, it can unload the module to free up memory and resources.

Modular applications have the following benefits:

- Smaller initial download size of the SWF file.
- Less memory use of overall application when modules are unloaded.
- Shorter load time due to smaller SWF file size.
- Better encapsulation of related aspects of an application. For example, a “reporting” feature can be separated into a module that you can then work on independently.

Modules and sub-applications are similar in many ways. Before deciding on an architecture for your applications, see the comparison of these two approaches in “Comparing loaded applications to modules” on page 187.

Benefits of modules

A module is a special type of dynamically loadable SWF file that contains an IFlexModuleFactory class factory. This allows an application to load code at run time and create class instances without requiring that the class implementations be linked into the main application.

Modules are similar to Runtime Shared Libraries (RSLs) in that they separate code from an application into separately loaded SWF files. Modules are very flexible because modules can be loaded and unloaded at run time and compiled without the application.

Two common scenarios in which using modules is beneficial are a large application with different user paths and a portal application.

An example of the first common scenario is an enormous insurance application that includes thousands of screens, for life insurance, car insurance, health insurance, dental insurance, travel insurance, and veterinary pet insurance.

By using a traditional approach to Rich Internet Application (RIA) design, you might build a monolithic application with a hierarchical tree of MXML classes. Memory use and start-up time for the application would be significant, and the SWF file size would grow with each new set of functionality.

When using this application, however, any user accesses only a subset of the screens. By refactoring the screens into small groups of modules that are loaded on demand, you can improve the perceived performance of the main application and reduce the memory use. Also, when the application is separated into modules, developers’ productivity may increase due to better encapsulation of design. When rebuilding the application, the developers also have to recompile only the single module instead of the entire application.

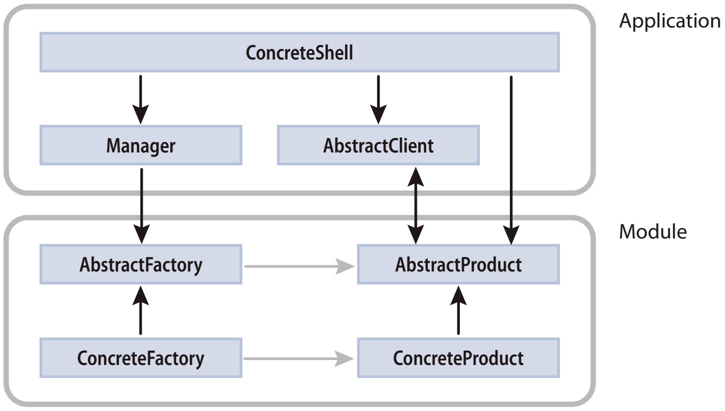
An example of the second common scenario is a system with a main portal application, written in ActionScript 3.0, that provides services for numerous portlets. Portlets are configured based on data that is downloaded on a per-user basis. By using the traditional approach, you might build an application that compiles in all known portlets. This is inefficient, both for deployment and development.

By using modules, you can establish an interface that contains portal services, and a generic portlet interface. You can use XML data to determine which modules to load for a given session. When the module is loaded, you obtain a handle to a class factory inside the module, and from that you create an instance of a class that implements the portlet interface. In this scenario, full recompilation is necessary only if the interfaces change.

Module API details

Modules implement a class factory with a standard interface. The product of that class factory implements an interface known to the shell, or the shell implements an interface known to the modules. These shared interfaces reduce hard dependencies between the shell and the module. This provides type-safe communication and enforces an abstraction layer without adding significantly to the SWF file size.

The following image shows the relationship between the shell and the module's interfaces:



The ModuleManager manages the set of loaded modules, which are treated as a map of Singletons that are indexed by the module URL. Loading a module triggers a series of events that let clients monitor the status of the module. Modules are only ever loaded once, but subsequent reloads also dispatch events so that client code can be simplified and rely on using the `READY` event to know that the module's class factory is available for use.

The ModuleLoader class is a thin layer on top of the ModuleManager API that is intended to act similarly to the mx.controls.SWFLoader class for modules that only define a single visual UIComponent. The ModuleLoader class is the easiest class to use when implementing a module-based architecture, but the ModuleManager provides greater control over the modules.

The ModuleLoader class implements the INavigatorContent interface so that it can be used directly by MX-based navigator containers (such as TabNavigator). The ModuleLoader class does not have any UI associated with it. All UI is defined by the module that it loads.

The Spark Module class extends the SkinnableContainer class. This means you can skin it and add visual components, including graphics, as children.

Module domains and sharing class libraries

By default, a module is loaded into a child domain of the current application domain. You can specify a different application domain by using the `applicationDomain` property of the ModuleLoader class.

Because a module is loaded into a child domain, it owns class definitions that are not in the main application's domain. For example, the first module to load the PopUpManager class becomes the owner of the PopUpManager class for the entire application because it registers the manager with the SingletonManager. If another module later tries to use the PopUpManager, Adobe ® Flash® Player throws an exception.

One solution is to use framework RSLs when compiling your applications and modules (RSLs are enabled by default). The definitions of the manager classes will be loaded in the framework RSL by the main application. Then, all the modules and sub-applications can share it. For more information about using framework RSLs with modules, see "Using RSLs with modules" on page 145.

If you do not use RSLs, the solution is to ensure that managers such as PopUpManager and DragManager and any other shared services are defined by the main application (or loaded late into the shell's application domain). When you promote one of those classes to the main application, the class can then be used by all modules. Typically, this is done by adding the following to a script block in the main application:

```
import mx.managers.PopUpManager;
import mx.managers.DragManager;
import mx.managers.ToolTipManager;
import mx.managers.CursorManager;
import mx.core.EmbeddedFontRegistry;

private var popUpManager:PopUpManager;
private var dragManager:DragManager;
private var tooltipManager:ToolTipManager;
private var cursorManager:CursorManager;
private var embeddedFontRegistry:EmbeddedFontRegistry;
```

You should only define these classes in your main application if your modules use the related functionality. For example, define the EmbeddedFontRegistry class in your main application if one or more of your modules uses embedded fonts.

This technique also applies to components. The module that first uses the component owns that component's class definition in its domain. As a result, if another module tries to use a component that has already been used by another module, its definition will not match the existing definition.

To avoid a mismatch of component definitions, create an instance of the component in the main application. The result is that the definition of the component is owned by the main application and can be used by modules in any child domain.

By default, modules do not share the main application's StyleManager, however. They have their own instances of the IStyleManager2 class. As a result, modules can define their own styles. For example, style properties set on a Button control in one module are not applied to the Button control in another module or to the main application.

Because a module must be in the same security domain as the application (SWF) that loads it, when you're using modules in an AIR application, any module SWF must be located in the same directory as the main application SWF or one of its subdirectories, which ensures that, like the main application SWF, the module SWF is in the AIR application security sandbox. One way to verify this is to ensure that a relative URL for the module's location doesn't require "../" ("up one level") notation to navigate outside the application directory or one of its subdirectories.

For more information about application domains, see "Developing and loading sub-applications" on page 176.

Create a modular application

To create a modular application, you create separate classes for each module, and an application that loads the modules. Adobe® Flash® Builder™ provides some mechanisms for making module use easier.

1 Create any number of modules. An MXML-based module file's root tag is `<s:Module>`. ActionScript-based modules extend either the Module or ModuleBase class.

2 Compile each module as if it were an application. You can do this by using the mxmlc command-line compiler or the compiler built into Adobe Flash Builder.

3 Create an Application class. This is typically an MXML file whose root tag is `<s:Application>`, but it can also be an ActionScript-only application.

4 In the Application file, use an `<s:ModuleLoader>` tag to load each of the modules. You can also load modules by using methods of the spark.modules.ModuleLoader and mx.modules.ModuleManager classes in ActionScript.

Using styles with modules

When you set styles on modules, the style properties are set on the local StyleManager. Each module has its own instance of the IStyleManager2 class. This means that each module can load its own set of styles, and its styles do not necessarily affect the styles of other modules.

After an application finishes loading a module, the module's styles are merged with the styles of the application. The module's StyleManager walks the chain of parent modules and applications, up to the top-level StyleManager, and merges its styles with those set on the StyleManagers above it.

If during a style merge, a module encounters a style that it already sets on itself, the style is ignored. If the module encounters a style not set on itself, the style is added to the merged style definition. The styles set on the StyleManager that is closest to the module wins.

The following example loads two modules. The main application sets the `color` and `cornerRadius` style properties on the Button control type selector. The modules each set the `color` property on the Button control type selector. The merged styles result in the Buttons having a corner radius of 10, with colors set by each module. This shows how style merges work.

```
<?xml version="1.0"?>
<!-- modules/StyleModLoaderApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            color:blue;
            cornerRadius:10;
        }
    </fx:Style>
    <s:VGroup>
        <s:Label id="l1" text="Module 1"/>
        <s:ModuleLoader id="ml1" url="mxmlmodules/StyleMod1.swf"/>
    </s:VGroup>
    <s:VGroup>
        <s:Label id="l2" text="Module 2"/>
        <s:ModuleLoader id="ml2" url="mxmlmodules/StyleMod2.swf"/>
    </s:VGroup>
    <s:Button id="myButton" label="Main App Button"/>
</s:Application>
```

Module 1:

```
<?xml version="1.0"?>
<!-- modules/mxmlmodules/StyleMod1.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            color:red;
        }
    </fx:Style>

    <s:Button label="StyleMod1"/>
</s:Module>
```

Module 2:

```
<?xml version="1.0"?>
<!-- modules/mxmlmodules/StyleMod2.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            color:green;
        }
    </fx:Style>

    <s:Button label="StyleMod2"/>
</s:Module>
```

To prevent style merges, set the `isolate-styles` compiler argument to `false`. By doing this, you might trigger type coercion errors when loading skins. Modules also might not be properly garbage collected when they are unloaded. This is because the main application's StyleManager will maintain references to the module even after is is unloaded. When you set `isolate-styes` to `false`, if more than one module loads a style, the first one loaded wins. In this case, styles set on modules can be overridden by those set on other modules.

In the previous example, if you set the `isolate-styles` compiler argument to `false`, the color of the Button controls' labels in both modules would be red, because that is definition that is first loaded.

The `getStyleDeclarations()` method returns only the local style definitions. To get the merged style definitions, you can use the `getMergedStyleDeclaration()` method. All methods that modify style definitions affect only the local style definitions and not the merged style definitions.

Style properties are merged when the module is loaded. This means that style properties set on the main application and all child applications and modules are set on the module if the module does not override that style. Merged styles are set on a per-property basis, starting with the closest StyleManager and working upwards to the top-level StyleManager.

Even when using merged styles, child modules still inherit their parent module or application's inheritable style properties when those settings are applied at runtime. Their StyleManager is not changed, but the values of the properties are inherited and applied where applicable. If you set a property at run time on the main application, the modules inherit that style immediately, as the following example shows:

```
<?xml version="1.0"?>
<!-- modules/StyleModLoaderApp2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        /* When you set this style in the main application, the modules immediately
           inherit the value. */
        private function changeStyle():void {
            styleManager.getStyleDeclaration("spark.components.Button").setStyle("fontSize",
15);
        }
    </fx:Script>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            color:blue;
            cornerRadius:10;
        }
    </fx:Style>
    <s:VGroup>
        <s:Label id="l1" text="Module 1"/>
        <s:ModuleLoader id="ml1" url="mxmlmodules/StyleMod1.swf"/>
    </s:VGroup>
    <s:VGroup>
        <s:Label id="l2" text="Module 2"/>
        <s:ModuleLoader id="ml2" url="mxmlmodules/StyleMod2.swf"/>
    </s:VGroup>
    <s:Button label="Change Styles" click="changeStyle()"/>
</s:Application>
```

For information on using run-time style sheets with modules, see “Using run-time style sheets with modules and sub-applications” on page 1554.

If you use run-time resource bundles with loaded modules, you should consider setting the `addResourceBundle()` method’s `useWeakReferences` parameter to `true`. For more information, see “Preventing memory leaks in modules and sub-applications” on page 2118.

Using RSLs with modules

Starting with Flex 4.5, modules work much more efficiently with RSLs. Modules will not load RSLs that the main application has already loaded, and modules can share RSLs with other modules.

The application only loads those framework RSLs that are needed at startup, and creates placeholders for all remaining framework RSLs. When a module is loaded, it does not try to load RSLs that are already loaded by the main application. If the module needs a framework RSL that is not initially loaded by the main application (and has a placeholder), then the module loads the RSL.

In addition, when a module loads an RSL, you can specify which domain the RSL is loaded into with the `application-domain` compiler argument. This lets you load an RSL into the parent, current, or top-level application domains. This applies to both framework RSLs and custom RLSs.

For more information about using modules and RSLs, see “Using RSLs with modules and sub-applications” on page 258.

Using pop-ups with modules

When using modules as pop-ups, you might not be able to click or drag the pop-up. The solution is to create a subclass of the pop-up (such as a TitleWindow container) that you use as the top-level MXML tag in the Module.

For more information, see Alex's Flex Closet.

Writing modules

Modules are classes just like application files. You can create them either in ActionScript or by extending a Flex class by using MXML tags. You can create modules in MXML and in ActionScript.

After you compile a module, you can load it into an application or another module. Typically, you use one of the following techniques to load MXML-based modules:

- ModuleLoader — The ModuleLoader class provides the highest-level API for handling modules. For more information, see “Using the ModuleLoader class to load modules” on page 150.
- ModuleManager — The ModuleManager class provides a lower-level API for handling modules than the ModuleLoader class does. For more information, see “Using the ModuleManager class to load modules” on page 152.

Creating MXML-based modules

To create a module in MXML, you extend the spark.modules.Module class by creating a file whose root tag is `<s:Module>`. In that tag, ensure that you add any namespaces that are used in that module. You must also include an XML type declaration tag at the beginning of the file, such as the following:

```
<?xml version="1.0"?>
```

The following example is a module that includes a Chart control:

```
<?xml version="1.0"?>
<!-- modules/ColumnChartModule.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%" >

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            {Month:"Jan", Profit:2000, Expenses:1500},
            {Month:"Feb", Profit:1000, Expenses:200},
            {Month:"Mar", Profit:1500, Expenses:500}
        ]);
    ]]></fx:Script>
    <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
</s:Module>
```

After you create a module, you compile it as if it were an application. For more information on compiling modules, see “Compiling modules” on page 148.

Creating ActionScript-based modules

To create a module in ActionScript, you can create a file that extends either the spark.modules.Module class or the mx.modules.ModuleBase class.

Extending the Module class is the same as using the `<s:Module>` tag in an MXML file. You should extend this class if your module interacts with the framework; this typically means that it adds objects to the display list or otherwise interacts with visible objects.

To see an example of an ActionScript class that extends the Module class, create an MXML file with the root tag of `<s:Module>`. When you compile this file, set the value of the `keep-generated-actionscript` compiler property to `true`. The Flex compiler stores the generated ActionScript class in a directory called generated. You will notice that this generated class contains code that you probably will not understand. As a result, you should not write ActionScript-based modules that extend the Module class; instead, you should use MXML to write such modules.

If your module does not include any framework code, you can create a class that extends ModuleBase. If you use the ModuleBase class, your module will typically be smaller than if you use a module based on the Module class because it does not have any framework class dependencies.

The following example creates a simple module that does not contain any framework code and therefore extends the ModuleBase class:

```
// modules/asmodules/SimpleModule.as
package {
    import mx.modules.ModuleBase;
    public class SimpleModule extends ModuleBase {
        public function SimpleModule() {
            trace("SimpleModule created");
        }

        public function computeAnswer(a:Number, b:Number):Number {
            return a + b;
        }
    }
}
```

To call the `computeAnswer()` method on the ActionScript module, you can use one of the techniques shown in "Accessing modules from the parent application" on page 173.

Compiling modules

The way you compile modules is similar to the way you compile applications. On the command line, you use the mxmlc command-line compiler; for example:

```
mxmlc MyModule.mxml
```

In Flash Builder, you compile all modules in a project when you compile the main application in that project.

The result of compiling a module is a SWF file that you load into your application. You cannot run the module-based SWF file as a stand-alone application or load it into a browser window. It must be loaded by an application as a module. Modules should not be opened directly by Adobe® Flash® Player or Adobe® AIR,™ or requested through a browser directly.

When you compile your module, you should try to remove redundancies between the module and the application that uses it. To do this on the command line, you create a link report for the application, and then externalize any assets in the module that appear in that report. Flash Builder can do this for you automatically. For more information, see "Reducing module size" on page 148.

Reducing module size

Module size varies based on the components and classes that are used in the module. By default, a module externalizes all framework code that its components depend on by using RSLs. However, other custom classes and libraries are not externalized by default, which can cause modules to be larger than necessary by linking classes that overlap with the application's classes.

To reduce the size of the modules, you can optimize the module by instructing the compiler to externalize classes that are included by the application. This includes custom classes and framework classes. The result is that the module includes only the classes it requires, while the framework code and other dependencies are included in the application.

By default, Flash Builder optimizes modules by externalizing classes against the default application in the module's project. If your project has multiple applications, you might need to change the application that a module is compiled against. Otherwise, if you try to load a module that was compiled against one application into another application, you could get run time errors claiming that classes are not found.

If you want to use a module with more than one application, you might want to optimize it for no applications so that it contains all the class references necessary to run, regardless of the parent application.

Change the application that a module is optimized for in Flash Builder

1 Select Project > Properties.

2 Select Flex Modules.

3 Select the module in the list and click the Edit button.

4 To change the application that a module is optimized for, select the Optimize For Application option, and select the new application from the drop down list.

To not optimize the module for any application, select the Do Not Optimize option.

Create and use a linker report with the command-line compiler

To externalize framework classes with the command-line compiler, you generate a linker report from the application that loads the modules. You then use this report as input to the module's `load-externs` compiler option. The compiler externalizes all classes from the module for which the application contains definitions. This process is also necessary if your modules are in a separate project from your main application in Flash Builder.

1 Generate the linker report and compile the application:

```
mxmlc -link-report=report.xml MyApplication.mxml
```

The default output location of the linker report is the same directory as the compiler. In this case, it would be in the bin directory.

2 Compile the module and pass the linker report to the `load-externs` option:

```
mxmlc -load-externs=report.xml MyModule.mxml
```

Recompiling modules

If you change a module, you do not have to recompile the application that uses the module if that module is in the same project. This is because the application loads the module at run time and does not check against it at compile time. Similarly, if you make changes to the application, you do not have to recompile the module. Just as the application does not check against the module at compile time, the module does not check against the application until run time.

If the module is in a separate project than the application that loads it, you must recompile the module separately.

However, if you make changes that might affect the linker report or common code, you should recompile both the application and the modules.

***Note:** If you externalize the module's dependencies by using the `load-externs` or `optimize` option, your module might not be compatible with future versions of Adobe Flex. You might be required to recompile the module. To ensure that a future application can use a module, compile that module with all the classes it requires. This also applies to applications that you load inside other applications.*

Debugging modules

To debug an application that uses modules, you set the `debug` compiler option to `true` for the modules when you compile them. Otherwise, you will not be able to set breakpoints in the modules or gather other debugging information from them. In Flash Builder, debugging is enabled by default. On the command line, debugging is disabled by default. You must also set the `debug` option to `true` when you compile the application that loads the modules that you want to debug.

A common issue that occurs when using multiple modules is that modules sometimes own the class definitions that the other modules want to use. Because they are in sibling application domains, the module that loaded the class definition first owns the definition for that class, but other modules will experience errors when they try to use that class. The solution is to promote the class definition to the main application domain so that all modules can use the class. For more information, see "Module domains and sharing class libraries" on page 141.

Loading and unloading modules

There are several techniques you can use to load and unload modules in your applications. These techniques include:

- ModuleLoader — The ModuleLoader class provides the highest-level API for handling modules. For more information, see "Using the ModuleLoader class to load modules" on page 150.
- ModuleManager — The ModuleManager class provides a lower-level API for handling modules than the ModuleLoader class does. For more information, see "Using the ModuleManager class to load modules" on page 152.

When you're using modules in an AIR application, the module SWF file must be located in the same directory as the main application SWF file or one of its subdirectories.

Using the ModuleLoader class to load modules

You can use the ModuleLoader class to load a module in an application or other module. The easiest way to do this in an MXML application is to use the `<s:ModuleLoader>` tag. You set the value of the `url` property to point to the location of the module's SWF file. The following example loads the module when the application first starts:

```
<?xml version="1.0"?>
<!-- modules/MySimplestModuleLoader.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:ModuleLoader url="ColumnChartModule.swf"/>
</s:Application>
```

You can change the timing of when the module loads by setting the value of the `url` property at some other time, such as in response to an event. Setting the target URL of a ModuleLoader object triggers a call to the `loadModule()` method. This occurs when you first create a ModuleLoader object with the `url` property set. It also occurs if you change the value of that property.

If you set the value of the `url` property to an empty string (`""`) or `null`, the ModuleLoader object unloads the current module by calling the `release()` method.

You can have multiple instances of the ModuleLoader class in a single application. The following example loads the modules when the user navigates to the appropriate tabs in the TabNavigator container:

```
<?xml version="1.0"?>
<!-- modules/URLModuleLoaderApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Panel title="Module Example" width="100%" height="100%">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>

        <mx:TabNavigator id="tn"
            paddingTop="10"
            paddingLeft="10"
            paddingRight="10"
            paddingBottom="10"
            width="100%" height="100%"
            creationPolicy="auto">
            <s:ModuleLoader id="ml1"
                label="ColumnChart Module"
                url="ColumnChartModule.swf"/>
            <s:ModuleLoader id="ml2"
                label="BarChart Module"
                url="BarChartModule.swf"/>
        </mx:TabNavigator>
    </s:Panel>
</s:Application>
```

You can also use the ModuleLoader API to load and unload modules with the `loadModule()` and `unloadModule()` methods. These methods take no parameters; the ModuleLoader class loads or unloads the module that matches the value of the current `url` property.

The following example loads and unloads the module when you click the button:

```
<?xml version="1.0"?>
<!-- modules/ASModuleLoaderApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import spark.modules.ModuleLoader;
        public function createModule(m:ModuleLoader, s:String):void {
            if (!m.url) {
                m.url = s;
            }
            m.loadModule();
        }

        public function removeModule(m:ModuleLoader):void {
            m.unloadModule();
        }
        ]]>
    </fx:Script>
    <s:Panel title="Module Example" width="100%" height="100%">
        <mx:TabNavigator id="tn"
            paddingTop="10"
            paddingLeft="10"
            paddingRight="10"
            paddingBottom="10"
            width="100%" height="100%"
            creationPolicy="auto">
            <s:NavigatorContent label="ColumnChartModule">
                <s:layout>
                    <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
                </s:layout>
                <s:Button label="Load"
                    click="createModule(chartModuleLoader, l1.text)"/>
                <s:Button label="Unload"
                    click="removeModule(chartModuleLoader)"/>
                <s:Label id="l1" text="ColumnChartModule.swf"/>
                <s:ModuleLoader id="chartModuleLoader"/>
            </s:NavigatorContent>
            <s:NavigatorContent label="FormModule">
                <s:layout>
                    <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
                </s:layout>
                <s:Button label="Load"
                    click="createModule(formModuleLoader, l2.text)"/>
                <s:Button label="Unload"
                    click="removeModule(formModuleLoader)"/>
                <s:Label id="l2" text="FormModule.swf"/>
                <s:ModuleLoader id="formModuleLoader"/>
            </s:NavigatorContent>
        </mx:TabNavigator>
    </s:Panel>
</s:Application>
```

When you load a module, Flex ensures that there is only one copy of a module loaded, no matter how many times you call the `load()` method for that module.

Using the ModuleManager class to load modules

You can use the ModuleManager class to load the module. This technique is less abstract than using the `<s:ModuleLoader>` tag, but it does provide you with greater control over how and when the module is loaded.

To use the ModuleManager to load a module in ActionScript:

1 Get a reference to the module's IModuleInfo interface by using the ModuleManager `getModule()` method.

2 Call the interface's `load()` method.

The application that loads the module should pass in its `moduleFactory` property. This lets the module know who its parent style manager is. When using the `load()` method, you can specify the application's `moduleFactory` with the fourth parameter, as the following example shows:

```
info.load(null, null, null, moduleFactory);
```

3 Use the `factory` property of the interface to call the `create()` method and cast the return value as the module's class. If you are adding the module to a container, you can cast the return value as an IVisualElement (for Spark containers) or a DisplayObject (for MX containers) so that they can be added to the display list.

The following example shell application loads the ColumnChartModule.swf file. The example then adds the modules to the display list so that it appears when the application starts:

```
<?xml version="1.0"?>
<!-- modules/ModuleLoaderApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.events.ModuleEvent;
        import mx.modules.ModuleManager;
        import mx.modules.IModuleInfo;
        import mx.core.IVisualElement;
        public var info:IModuleInfo;

        private function initApp():void {
            info = ModuleManager.getModule("ColumnChartModule.swf");
            info.addEventListener(ModuleEvent.READY, modEventHandler);
            /* Load the module into memory. Calling load() makes the
               IFlexModuleFactory available. You can then get an
               instance of the class using the factory's create()
               method. */
            info.load(null, null, null, moduleFactory);
        }
        /* Add an instance of the module's class to the display list. */
        private function modEventHandler(e:ModuleEvent):void {
            /* For MX containers, cast to a DisplayObject. */
            vb1.addChild(info.factory.create() as DisplayObject);
            /* For Spark containers, cast to a UIComponent. */
            vg1.addElement(info.factory.create() as IVisualElement);
        }
        ]]>
    </fx:Script>
    <!-- MX container -->
    <mx:VBox id="vb1">
        <s:Label text="Module loaded in MX VBox container:"/>
    </mx:VBox>
    <!-- Spark container -->
    <s:VGroup id="vg1">
        <s:Label text="Module loaded in Spark VGroup container:"/>
    </s:VGroup>
</s:Application>
```

The IModuleInfo class's `load()` method also optionally takes an ApplicationDomain and a SecurityDomain as arguments. If you do not specify either of these (or set them to `null`), then the module is loaded into a new child domain.

MXML-based modules can load other modules. Those modules can load other modules, and so on.

Be sure to define the module instance outside of a function, so that it is not in the function's local scope. Otherwise, the object might be garbage collected and the associated event listeners might never be invoked.

If you remove all references to the module, it will be garbage collected. You do not need to call the `unload()` method when adding and removing modules using the IModuleInfo class. You just need to set the IModuleInfo instance to `null`.

Loading modules from different servers

To load a module from one server into an application running on a different server, you must establish trust between the module and the application that loads it.

Access applications across domains

1 In your loading application, you must call the `allowDomain()` method and specify the target domain from which you load a module. So, specify the target domain in the preinitialize event handler of your application to ensure that the application is set up before the module is loaded.

2 In the cross-domain file of the remote server where your module is, add an entry that specifies the server on which the loading application is running.

3 Load the cross-domain file on the remote server in the preinitialize event handler of your loading application.

4 In the loaded module, call the `allowDomain()` method so that it can communicate with the loader.

The following example shows the `init()` method of the loading application:

```
public function setup():void {
    Security.allowDomain("remoteservername");
    Security.loadPolicyFile("http://remoteservername/crossdomain.xml");
    var request:URLRequest = new URLRequest("http://remoteservername/crossdomain.xml");
    var loader:URLLoader = new URLLoader();
    loader.load(request);
}
```

The following example shows the loaded module's `init()` method:

```
public function initMod():void {
    Security.allowDomain("loaderservername");
}
```

The following example shows the cross-domain file that resides on the remote server:

```
<!-- crossdomain.xml file located at the root of the server -->
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all"/>
    <allow-access-from domain="loaderservername" to-ports="*"/>
</cross-domain-policy>
```

For more information about using the cross-domain policy file, see "Security" on page 117.

Preloading modules

When you first start an application that uses modules, the application's file size should be smaller than a similar application that does not use modules. As a result, there should be a reduction in wait time because the application can be loaded into memory and run before the modules' SWF files are even transferred across the network. However, there will be a delay when the user navigates to a part in the application that uses the module. This is because the modules are not by default preloaded, but rather loaded when they are first requested.

When a module is loaded by the application for the first time, the module's SWF file is transferred across the network and stored in the browser's cache. If the application unloads that module, but then later reloads it, there should be less wait time because Flash Player loads the module from the cache rather than across the network.

Module SWF files, like all SWF files, reside in the browser's cache unless and until a user clears them. As a result, modules can be loaded by the main application across several sessions, reducing load time; but this depends on how frequently the browser's cache is flushed.

You can preload modules at any time so that you can have the modules' SWF files in memory even if the module is not currently being used.

To preload modules on application startup, use the IModuleInfo class `load()` method. This loads the module into memory but does not create an instance of the module.

The following example loads the BarChartModule.swf module when the application starts up, even though it will not be displayed until the user navigates to the second pane of the TabNavigator container. Without preloading, the user would wait for the SWF file to be transferred across the network when they navigated to the second pane of the TabNavigator.

```
<?xml version="1.0"?>
<!-- modules/PreloadModulesApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="preloadModules()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
        import mx.events.ModuleEvent;
        import mx.modules.ModuleManager;
        import mx.modules.IModuleInfo;

        private function preloadModules():void {
            /* Get a reference to the module's interface. */
            var info:IModuleInfo =
                ModuleManager.getModule("BarChartModule.swf");
            info.addEventListener(ModuleEvent.READY, modEventHandler);

            /* Load the module into memory. The module will be
               displayed when the user navigates to the second
               tab of the TabNavigator. */
            info.load();
        }
        private function modEventHandler(e:ModuleEvent):void {
            trace("module event: " + e.type); // "ready"
        }
        ]]>
    </fx:Script>
    <s:Panel title="Module Example" width="100%" height="100%">
        <mx:TabNavigator id="tn"
            paddingTop="10"
            paddingLeft="10"
            paddingRight="10"
            paddingBottom="10"
            width="100%" height="100%"
            creationPolicy="auto">
            <s:ModuleLoader label="ColumnChartModule"
                url="ColumnChartModule.swf"/>
            <s:ModuleLoader label="BarChartModule"
                url="BarChartModule.swf"/>
        </mx:TabNavigator>
    </s:Panel>
</s:Application>
```

Using ModuleLoader events

The ModuleLoader class triggers several events, including `setup`, `ready`, `loading`, `unload`, `progress`, `error`, and `urlChanged`. You can use these events to track the loading process, and find out when a module has been unloaded or when the ModuleLoader target URL has changed.

The following example uses a custom ModuleLoader component. This component reports all the events of the modules as they are loaded by the main application.

Custom ModuleLoader:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- modules/CustomModuleLoader.mxml -->
<s:ModuleLoader
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns="*"
    creationComplete="init()">

  <fx:Script>
    <![CDATA[
        import mx.core.UIComponent;

        public var standin:UIComponent;
        public function init():void {
            addEventListener("urlChanged", onUrlChanged);
            addEventListener("loading", onLoading);
            addEventListener("progress", onProgress);
            addEventListener("setup", onSetup);
            addEventListener("ready", onReady);
            addEventListener("error", onError);
            addEventListener("unload", onUnload);
            standin = panel;
            removeElement(standin);
        }
```

```
public function onUrlChanged(event:Event):void {
    if (url == null) {
        if (contains(standin))
            removeElement(standin);
    } else {
        if (!contains(standin))
            addElement(standin);
    }
    progress.indeterminate=true;
    unload.enabled=false;
    reload.enabled=false;
}
public function onLoading(event:Event):void {
    progress.label="Loading module " + url;
    if (!contains(standin))
        addElement(standin);
    progress.indeterminate=true;
    unload.enabled=false;
    reload.enabled=false;
}
public function onProgress(event:Event):void {
    progress.label="Loaded %1 of %2 bytes...";
    progress.indeterminate=false;
    unload.enabled=true;
    reload.enabled=false;
}
public function onSetup(event:Event):void {
    progress.label="Module " + url + " initialized!";
    progress.indeterminate=false;
    unload.enabled=true;
    reload.enabled=true;
}
public function onReady(event:Event):void {
    progress.label="Module " + url + " successfully loaded!";
    unload.enabled=true;
    reload.enabled=true;
    if (contains(standin))
        removeElement(standin);
}
public function onError(event:Event):void {
    progress.label="Error loading module " + url;
    unload.enabled=false;
    reload.enabled=true;
}
public function onUnload(event:Event):void {
    if (url == null) {
        if (contains(standin))
            removeElement(standin);
    } else {
        if (!contains(standin))
```

```
                    addElement(standin);
            }
            progress.indeterminate=true;
            progress.label="Module " + url + " was unloaded!";
            unload.enabled=false;
            reload.enabled=true;
        }
    ]]>
  </fx:Script>
  <s:Panel id="panel" width="100%" title="Status &amp; Operations">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:ProgressBar width="100%" id="progress" source="{this}"/>
    <s:HGroup width="100%">
      <s:Button id="unload" label="Unload Module" click="unloadModule()"/>
      <s:Button id="reload" label="Reload Module" click="unloadModule();loadModule()"/>
    </s:HGroup>
  </s:Panel>
</s:ModuleLoader>
```

Main application:

```
<?xml version="1.0"?>
<!-- modules/EventApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns="*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var selectedModule:Object;
        ]]>
    </fx:Script>
    <s:ComboBox width="215" labelField="label" prompt="Select Coverage"
        close="selectedModule=ComboBox(event.target).selectedItem">
        <s:dataProvider>
            <s:ArrayCollection>
```

```
                <fx:Object label="Life Insurance"
                    module="insurancemodules/LifeInsurance.swf"/>
                <fx:Object label="Auto Insurance"
                    module="insurancemodules/AutoInsurance.swf"/>
                <fx:Object label="Home Insurance"
                    module="insurancemodules/HomeInsurance.swf"/>
            </s:ArrayCollection>
        </s:dataProvider>
    </s:ComboBox>
    <s:Panel width="100%" height="100%" title="Custom Module Loader">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <CustomModuleLoader id="mod" width="100%" url="{selectedModule.module}"/>
    </s:Panel>
    <s:HGroup>
        <s:Button label="Unload" click="mod.unloadModule()"/>
        <s:Button label="Nullify" click="mod.url=null"/>
    </s:HGroup>
</s:Application>
```

The insurance modules used in this example are simple forms, such as the following:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- modules/insurancemodules/AutoInsurance.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    backgroundColor="#ffffff"
    width="100%" height="100%">
    <s:Label x="147" y="50" text="Auto Insurance"
        fontSize="28" fontFamily="Myriad Pro"/>
    <s:Form left="47" top="80">
        <s:FormHeading label="Coverage"/>
        <s:FormItem label="Latte Spillage">
            <s:TextInput id="latte" width="200"/>
        </s:FormItem>
        <s:FormItem label="Shopping Cart to the Door">
            <s:TextInput id="cart" width="200"/>
        </s:FormItem>
        <s:FormItem label="Irate Moose">
            <s:TextInput id="moose" width="200"/>
        </s:FormItem>
        <s:FormItem label="Color Fade">
            <mx:ColorPicker/>
        </s:FormItem>
    </s:Form>
</s:Module>
```

Using the error event

The `error` event gives you an opportunity to gracefully fail when a module does not load for some reason. In the following example, you can load and unload a module by using the Button controls. To trigger an `error` event, change the URL in the TextInput control to a module that does not exist. The error handler displays a message to the user and writes the error message to the trace log.

```
<?xml version="1.0"?>
<!-- modules/ErrorEventHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout
                paddingTop="10"
                paddingLeft="10"
                paddingRight="10"
                paddingBottom="10"/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.events.ModuleEvent;
        import spark.modules.ModuleLoader;
        import spark.modules.Module;
        import mx.controls.Alert;

        private function errorHandler(e:ModuleEvent):void {
            Alert.show("There was an error loading the module." +
                " Please contact the Help Desk.\n" +
                e.errorText);
        }

        public function createModule():void {
            if (chartModuleLoader.url == ti1.text) {
                /* If they are the same, call the loadModule() method. */
                chartModuleLoader.loadModule();
            } else {
                /* If they are not the same, then change the url,
                   which then triggers a call to the loadModule() method. */
                chartModuleLoader.url = ti1.text;
            }
        }

        public function removeModule():void {
            chartModuleLoader.unloadModule();
        }
        ]]>
    </fx:Script>
    <s:Panel title="Module Example" height="90%" width="90%">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:HGroup>
            <s:Label text="URL:"/>
            <s:TextInput width="200" id="ti1" text="ColumnChartModule.swf"/>
            <s:Button label="Load" click="createModule()"/>
            <s:Button label="Unload" click="removeModule()"/>
        </s:HGroup>
        <s:ModuleLoader id="chartModuleLoader" error="errorHandler(event)"/>
    </s:Panel>
</s:Application>
```

Using the progress event

You can use the progress event to track the progress of a module as it loads. When you add a listener for the progress event, Flex calls that listener at regular intervals during the module's loading process. Each time the listener is called, you can look at the bytesLoaded property of the event. You can compare this to the bytesTotal property to get a percentage of completion.

The following example reports the level of completion during the module's loading process. It also produces a simple progress bar that shows users how close the loading is to being complete.

```
<?xml version="1.0"?>
<!-- modules/SimpleProgressEventHandler.mxml -->
<s:Application
    creationComplete="initApp()"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.events.ModuleEvent;
        import flash.events.ProgressEvent;
        import spark.modules.Module;
        import spark.modules.ModuleLoader;

        [Bindable]
        public var progBar:String = "";
        [Bindable]
        public var progMessage:String = "";

        private function progressEventHandler(e:ProgressEvent):void {
            progBar += ".";
            progMessage =
                "Module " +
                Math.round((e.bytesLoaded/e.bytesTotal) * 100) +
                "% loaded";
        }
        public function initApp():void {
            chartModuleLoader.url = "ColumnChartModule.swf";
        }
        public function createModule():void {
            chartModuleLoader.loadModule();
        }

        public function removeModule():void {
            chartModuleLoader.unloadModule();
            progBar = "";
            progMessage = "";
        }
        ]]>
    </fx:Script>
    <s:Panel title="Module Example" height="90%" width="90%">
        <s:layout>
            <s:VerticalLayout
                paddingTop="10" paddingLeft="10"
                paddingRight="10" paddingBottom="10"/>
        </s:layout>
        <s:HGroup>
            <s:Label id="l2" text="{progMessage}"/>
            <s:Label id="l1" text="{progBar}"/>
        </s:HGroup>
        <s:Button label="Load" click="createModule()"/>
        <s:Button label="Unload" click="removeModule()"/>
        <s:ModuleLoader id="chartModuleLoader"
            progress="progressEventHandler(event)"/>
    </s:Panel>
</s:Application>
```

You can also connect a module loader to a ProgressBar control. The following example creates a custom component for the ModuleLoader that includes a ProgressBar control. The ProgressBar control displays the progress of the module loading.

```
<?xml version="1.0"?>
<!-- modules/MySimpleModuleLoader.mxml -->
<s:ModuleLoader
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function clickHandler():void {
                if (!url) {
                    url="ColumnChartModule.swf";
                }
                loadModule();
            }
        ]]>
    </fx:Script>
    <mx:ProgressBar id="progress" width="100%" source="{this}"/>
    <s:HGroup width="100%">
      <s:Button id="load" label="Load" click="clickHandler()"/>
      <s:Button id="unload" label="Unload" click="unloadModule()"/>
      <s:Button id="reload" label="Reload" click="unloadModule();loadModule();"/>
    </s:HGroup>
</s:ModuleLoader>
```

You can use this module in a simple application, as the following example shows:

```
<?xml version="1.0"?>
<!-- modules/ComplexProgressEventHandler.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:local="*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Module Example" height="90%" width="90%">
        <s:layout>
            <s:VerticalLayout
                paddingTop="10" paddingLeft="10"
                paddingRight="10" paddingBottom="10"/>
        </s:layout>
        <s:Label text="Use the buttons below to load and unload the module."/>
        <local:MySimpleModuleLoader id="customLoader"/>
    </s:Panel>

</s:Application>
```

This example does not change the ProgressBar `label` property for all events. For example, if you load and then unload the module, the `label` property remains at "LOADING 100%". To adjust the label properly, you must define other event handlers for the ModuleLoader events, such as `unload` and `error`.

Passing data to modules

Communication between modules and the parent application, and among modules, is possible. You can use the following approaches to facilitate inter-module, application-to-module, and module-to-application communication:

- Interfaces — You can create ActionScript interfaces that define the methods and properties that modules and applications can access. This gives you greater control over module and application interaction. It also prevents you from creating dependencies between modules and applications. For more information, see “Using interfaces for module communication” on page 164.
- Query string parameters — Modules are loaded with a URL; you can pass parameters on this URL and then parse those parameters in the module. This is only a way to pass simple data, and is not appropriate for complex data. For more information, see “Passing data to modules with the query string” on page 166.
- ModuleLoader`'s child`, ModuleManager ‘s `factory`, and Application’s `parentApplication` properties — You can use these properties to access modules and applications. However, by using these properties, you create a tightly-coupled design that prevents code reuse and is easily broken. In addition, you also create dependencies among modules and applications that cause class sizes to be bigger. For more information, see “Accessing modules from the parent application” on page 173, “Accessing the parent application from modules” on page 169, and “Accessing modules from other modules” on page 171.

The following techniques for accessing methods and properties apply to parent applications as well as modules. Modules can load other modules, which makes the loading module similar to the parent application in the simpler examples.

Using interfaces for module communication

You can use an interface to provide module-to-application communication. Your modules implement the interface and your application calls its methods or sets its properties. The interface defines stubs for the methods and properties that you want the application and module to share. The module implements an interface known to the application, or the application implements an interface known to the module. This lets you avoid so-called hard dependencies between the module and the application.

In the main application, when you want to call methods on the module, you cast the ModuleLoader class's `child` property to an instance of the custom interface.

The following example application lets you customize the appearance of the module that it loads by calling methods on the custom IModuleInterface interface. The application also calls the `getModuleName()` method. This method returns a value from the module and sets a local property to that value.

```
<?xml version="1.0"?>
<!-- modules/interfaceexample/MainModuleApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns="*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.ModuleEvent;
            import mx.modules.ModuleManager;

            [Bindable]
            public var selectedItem:Object;
            [Bindable]
            public var currentModuleName:String;

            private function applyModuleSettings(e:Event):void {
                /* Cast the ModuleLoader's child to the interface.
                   This child is an instance of the module.
                   You can now call methods on that instance. */
                var ichild:* = mod.child as IModuleInterface;
                if (mod.child != null) {
                    /* Call setters in the module to adjust its
                       appearance when it loads. */
                    ichild.setAdjusterID(myId.text);
                    ichild.setBackgroundColor(myColor.selectedColor);
                } else {
                    trace("Uh oh. The mod.child property is null");
                }
                /* Set the value of a local variable by calling a method
                   on the interface. */
                currentModuleName = ichild.getModuleName();
            }

            private function reloadModule():void {
                // Reset the ColorPicker control:
                myColor.selectedColor = 0xFFFFFF;
                // Reload the module:
                mod.unloadModule();
                mod.loadModule();
            }
        ]]>
    </fx:Script>

    <s:Form>
        <s:FormItem label="Current Module:">
            <s:Label id="l1" text="{currentModuleName}"/>
        </s:FormItem>
        <s:FormItem label="Adjuster ID:">
            <s:TextInput id="myId" text="Enter your ID"/>
        </s:FormItem>
        <s:FormItem label="Background Color:">
            <mx:ColorPicker id="myColor"
                selectedColor="0xFFFFFF"
                change="applyModuleSettings(event)"/>
        </s:FormItem>
    </s:Form>

    <s:Label text="Long Shot Insurance" fontSize="24"/>
    <s:ComboBox labelField="label" prompt="Select Module"
        close="selectedItem=ComboBox(event.target).selectedItem">
        <s:dataProvider>
            <s:ArrayList>
                <fx:Object label="Auto Insurance" module="AutoInsurance2.swf"/>
            </s:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
    <s:Panel width="100%" height="100%">
        <s:ModuleLoader id="mod"
            width="80%" height="80%"
            url="{selectedItem.module}"
            ready="applyModuleSettings(event)"/>
    </s:Panel>
    <s:Button id="b1" label="Reload Module" click="reloadModule()"/>
</s:Application>
```

The following example defines a simple interface that has two getters and one setter. This interface is used by the application in the previous example.

```
// modules/interfaceexample/IModuleInterface.as
package
{
    import flash.events.IEventDispatcher;
    public interface IModuleInterface extends IEventDispatcher {

        function getModuleName():String;
        function setAdjusterID(s:String):void;
        function setBackgroundColor(n:Number):void;
    }
}
```

The following example defines the module that is loaded by the previous example. It implements the custom IModuleInterface interface.

```
<?xml version="1.0"?>
<!-- modules/interfaceexample/AutoInsurance2.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%" implements="IModuleInterface">

    <s:Panel id="p1" title="Auto Insurance"
        width="100%" height="100%"
        backgroundColor="{bgcolor}">
        <s:Label id="myLabel" text="ID: {adjuster}"/>
    </s:Panel>
    <fx:Script>
        <![CDATA[
            [Bindable]
            private var adjuster:String;
            [Bindable]
            private var bgcolor:Number;
            public function setAdjusterID(s:String):void {
                adjuster = s;
            }

            public function setBackgroundColor(n:Number):void {
                /* Use a bindable property to set values of controls
                   in the module. This ensures that the property will be set
                   even if Flex applies the property after the module is
                   loaded but before it is rendered by the player. */
                bgcolor = n;

                /* Don't do this. The backgroundColor style might not be set
                   by the time the ModuleLoader triggers the READY
                   event: */
                // p1.setStyle("backgroundColor", n);
            }

            public function getModuleName():String {
                return "Auto Insurance";
            }
        ]]>
    </fx:Script>
</s:Module>
```

In general, if you want to set properties on controls in the module by using external values, you should create variables that are bindable. You then set the values of those variables in the interface's implemented methods. If you try to set properties of the module's controls directly by using external values, the controls might not be instantiated by the time the module is loaded and the attempt to set the properties might fail.

Passing data to modules with the query string

One way to pass data to a module is to append query string parameters to the URL that you use to load the module. You can then parse the query string by using ActionScript to access the data.

In the module, you can access the URL by using the `loaderInfo` property. This property points to the LoaderInfo object of the loading SWF (in this case, the main application). The information provided by the LoaderInfo object includes load progress, the URLs of the loader and loaded content, the file size of the application, and the height and width of the application.

The following example application builds a unique query string for the module that it loads. The query string includes a `firstName` and `lastName` parameter.

```
<?xml version="1.0"?>
<!-- modules/QueryStringApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="500" width="400">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function initModule():void {
                // Build query string so that it looks something like this:
                // "QueryStringModule.swf?firstName=Nick&lastName=Danger"
                var s:String = "QueryStringModule.swf?" + "firstName=" +
                    ti1.text + "&lastName=" + ti2.text;
                // Changing the url property of the ModuleLoader causes
                // the ModuleLoader to load a new module.
                m1.url = s;
            }
        ]]>
    </fx:Script>
    <s:Form>
        <s:FormItem id="fi1" label="First Name:">
            <s:TextInput id="ti1"/>
        </s:FormItem>
        <s:FormItem id="fi2" label="Last Name:">
            <s:TextInput id="ti2"/>
        </s:FormItem>
    </s:Form>
    <s:ModuleLoader id="m1"/>

    <s:Button id="b1" label="Submit" click="initModule()"/>
</s:Application>
```

The following example module parses the query string that was used to load it. If the `firstName` and `lastName` parameters are set, the module prints the results in a TextArea. The module also traces some additional information available through the LoaderInfo object:

```
<?xml version="1.0"?>
<!-- modules/QueryStringModule.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="parseString()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.utils.*;
            [Bindable]
            private var salutation:String;

            public var o:Object = {};

            public function parseString():void {
                try {
                    /* Remove everything before the question mark, including
                       the question mark. */
                    var myPattern:RegExp = /.*\?/;
                    var s:String = this.loaderInfo.url.toString();
                    s = s.replace(myPattern, "");
                    /* Create an Array of name=value Strings. */
                    var params:Array = s.split("&");

                    /* Print the params that are in the Array. */
                    var keyStr:String;
                    var valueStr:String;
                    var paramObj:Object = params;
                    for (keyStr in paramObj) {
                        valueStr = String(paramObj[keyStr]);
                        ta1.text += keyStr + ":" + valueStr + "\n";
                    }
                    /* Set the values of the salutation. */
                    for (var i:int = 0; i < params.length; i++) {
                        var tempA:Array = params[i].split("=");
                        if (tempA[0] == "firstName") {
                            o.firstName = tempA[1];
                        }
                        if (tempA[0] == "lastName") {
                            o.lastName = tempA[1];
                        }
                    }
                    if (StringUtil.trim(o.firstName) != "" &&
                        StringUtil.trim(o.lastName) != "") {
                            salutation = "Welcome " +
                            o.firstName + " " + o.lastName + "!";
                    } else {
                        salutation = "Full name not entered."
                    }
                } catch (e:Error) {
                    trace(e);
                }
                /* Show some of the information available through loaderInfo: */
               ta2.text = "AS version: " + this.loaderInfo.actionScriptVersion;
               ta2.text += "\nApp height: " + this.loaderInfo.height;
               ta2.text += "\nApp width: " + this.loaderInfo.width;
               ta2.text += "\nApp bytes: " + this.loaderInfo.bytesTotal;
            }
        ]]>
    </fx:Script>
    <s:Label text="{salutation}"/>
    <s:TextArea height="75" width="250" id="ta1"/>
    <s:TextArea height="200" width="250" id="ta2"/>
</s:Module>
```

This example uses methods of the String and StringUtil classes, plus a for-in loop to parse the URLs. You can also use methods of the URLUtil and URLVariables classes to do this.

Modules are cached by their URL, including the query string. As a result, you will load a new module if you change the URL or any of the query string parameters in the URL. This can be useful if you want multiple instances of a module based on the parameters that you pass in the URL with the ModuleLoader.

Accessing the parent application from modules

Modules can access properties and methods of the parent application by using a reference to the `parentApplication` property. In most cases, you should avoid doing this as it creates a close coupling between the module and the application. Having this coupling directly negates some of the benefits of using modules.

The following example accesses the `expenses` property of the parent application when the module first loads. The module then uses this property, an ArrayCollection, as the source for its chart's data. When the user clicks the button, the module calls the `getNewData()` method of the parent application that returns a new ArrayCollection for the chart:

```
<?xml version="1.0"?>
<!-- modules/ChartChildModule.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%"
    creationComplete="getDataFromParent()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            private var expenses:ArrayCollection;
            // Access properties of the parent application.
            private function getDataFromParent():void {
                expenses = parentApplication.expenses;
            }
        ]]>
    </fx:Script>

    <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries xField="Month" yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries xField="Month" yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

     <s:Button id="b1"
        click="expenses = parentApplication.getNewData();"
        label="Get New Data"/>

</s:Module>
```

The following example shows the parent application that the previous example module uses:

```
<?xml version="1.0"?>
<!-- modules/ChartChildModuleLoader.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            public var expenses:ArrayCollection = new ArrayCollection([
                {Month:"Jan", Profit:2000, Expenses:1500},
                {Month:"Feb", Profit:1000, Expenses:200},
                {Month:"Mar", Profit:1500, Expenses:500}
            ]);
            public function getNewData():ArrayCollection {
                return new ArrayCollection([
                    {Month:"Apr", Profit:1000, Expenses:1100},
                    {Month:"May", Profit:1300, Expenses:500},
                    {Month:"Jun", Profit:1200, Expenses:600}
                ]);
            }
        ]]>
    </fx:Script>

    <s:ModuleLoader url="ChartChildModule.swf" id="m1"/>
</s:Application>
```

You can also call methods and access properties on other modules. For more information, see "Accessing modules from other modules" on page 171.

The drawback to this approach is that it can create dependencies on the parent application inside the module. In addition, the modules are no longer portable across multiple applications unless you ensure that you replicate the behavior of the applications.

To avoid these drawbacks, you should use interfaces that secure a contract between the application and its modules. This contract defines the methods and properties that you can access. Having an interface lets you reuse the application and modules as long as you keep the interface updated. For more information, see "Using interfaces for module communication" on page 164.

Accessing modules from other modules

You can access properties and methods of other modules by using references to the other modules through the parent application. You do this by using the ModuleLoader class's `child` property. This property points to an instance of the module's class, which lets you call methods and access properties. In most cases, you should avoid doing this as it creates a close coupling among the modules. Having this coupling directly negates some of the benefits of using modules in the first place.

The following example defines a single application that loads two modules. The InterModule1 module defines a method that returns a String. The InterModule2 module calls that method and sets the value of its Label to the return value of that method.

Main application:

```
<?xml version="1.0"?>
<!-- modules/InterModuleLoader.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        ]]>
    </fx:Script>

    <s:ModuleLoader url="InterModule1.swf" id="m1"/>

    <s:ModuleLoader url="InterModule2.swf" id="m2"/>
</s:Application>
```

Module 1:

```
<?xml version="1.0"?>
<!-- modules/InterModule1.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%">

    <fx:Script>
        <![CDATA[
            /* Defines the method that the other module calls. */
            public function getNewTitle():String {
                return "New Title";
            }
        ]]>
    </fx:Script>
</s:Module>
```

Module 2:

```
<?xml version="1.0"?>
<!-- modules/InterModule2.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            [Bindable]
            private var title:String = "Original Title";
            // Call method of another module.
            private function changeTitle():void {
                title = parentApplication.m1.child.getNewTitle();
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:Label id="l1" text="Title: "/>
        <s:Label id="myTitle" text="{title}"/>
    </s:HGroup>
    <s:Button id="b1" label="Change Title" click="changeTitle()"/>
</s:Module>
```

The application in this example lets the two modules communicate with each other. You could, however, define methods and properties on the application that the modules could access. For more information, see "Accessing the parent application from modules" on page 169.

As with accessing the parent application's properties and methods directly, using the technique described in this section can make your modules difficult to reuse and also can create dependencies that can cause the module to be larger than necessary. Instead, you should use interfaces to define the contract between modules. For more information, see "Using interfaces for module communication" on page 164.

Accessing modules from the parent application

You can access the methods and properties of a module from its parent application by getting an instance of the module's class. Referencing a module by its class name in an application causes the whole module and all of its dependencies to be linked into the application. This defeats the purpose of using modules. However, using this technique can be useful for debugging and testing.

You should only use interfaces to access the methods and properties of a module unless you want to create these dependencies. For more information, see "Using interfaces for module communication" on page 164.

If you use the ModuleLoader to load the module, you can call methods on a module from the parent application by referencing the ModuleLoader class's `child` property, and casting it to the module's class. The `child` property is an instance of the module's class. In this case, the module's class is the name of the MXML file that defines the module.

The following example calls the module's `getTitle()` method from the parent application:

Parent Application:

```
<?xml version="1.0"?>
<!-- modules/ParentApplication.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        [Bindable]
        private var s:String;

        private function getTitle():void {
            s = (m1.child as ChildModule1).getModTitle();
        }
    ]]></fx:Script>

    <s:Label id="l1" text="{s}"/>
    <s:ModuleLoader id="m1"
        url="ChildModule1.swf"
        ready="getTitle()"/>
</s:Application>
```

Module:

```
<?xml version="1.0"?>
<!-- modules/ChildModule1.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%">
    <fx:Script><![CDATA[
        /* Defines the method that the application calls. */
        public function getModTitle():String {
            return "Child Module 1";
        }
    ]]></fx:Script>
</s:Module>
```

If you load the module that you want to call by using the ModuleManager API, there is some additional coding in the shell application. You use the ModuleManager `factory` property to get an instance of the module's class. You can then call the module's method on that instance.

The following module example defines a single method, `computeAnswer()`:

```
<?xml version="1.0"?>
<!-- modules/mxmlmodules/SimpleModule.mxml -->
<s:Module
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            public function computeAnswer(a:Number, b:Number):Number {
                return a + b;
            }
        ]]>
    </fx:Script>
</s:Module>
```

The following example gets an instance of the SimpleModule class by using the `factory` property to call the `create()` method. It then calls the module’s `computeAnswer()` method on that instance:

```
<?xml version="1.0"?>
<!-- modules/mxmlmodules/SimpleMXMLApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
        import mx.modules.IModuleInfo;
        import mx.modules.ModuleManager;
        public var assetModule:IModuleInfo;
        public var sm:Object;

        [Bindable]
        public var answer:Number = 0;
        public function initApp():void {
            /* Get the IModuleInfo interface for the specified URL. */
            assetModule = ModuleManager.getModule("SimpleModule.swf");
            assetModule.addEventListener("ready", getModuleInstance);
            assetModule.load(null, null, null, moduleFactory);
        }
        public function getModuleInstance(e:Event):void {
            /* Get an instance of the module. */
            sm = assetModule.factory.create() as SimpleModule;
        }

        public function addNumbers():void {
            var a:Number = Number(ti1.text);
            var b:Number = Number(ti2.text);
            /* Call a method on the module. */
            answer = sm.computeAnswer(a, b).toString();
        }
    ]]>
  </fx:Script>
   <s:Form>
        <s:FormHeading label="Enter values to sum."/>
        <s:FormItem label="First Number">
            <s:TextInput id="ti1" width="50"/>
        </s:FormItem>
        <s:FormItem label="Second Number">
            <s:TextInput id="ti2" width="50"/>
        </s:FormItem>
        <s:FormItem label="Result">
            <s:Label id="ti3" width="100" text="{answer}"/>
        </s:FormItem>
        <s:Button id="b1" label="Compute" click="addNumbers()"/>
    </s:Form>
</s:Application>
```

In this example, you should actually create a module that extends the ModuleBase class in ActionScript rather than an MXML-based module that extends the Module class. This is because the example module does not have any visual elements and contains only a single method that computes and returns a value. A module that extends the ModuleBase class would be more lightweight than a class that extends Module. For more information on writing ActionScript-based modules that extend the ModuleBase class, see “Creating ActionScript-based modules” on page 147.

Developing and loading sub-applications

About loading sub-applications

Flex lets you load and unload sub-applications in a main application. Reasons to use sub-applications as part of your overall application architecture include the following:

- Reduce the size of the main application
- Encapsulate related functionality into a sub-application
- Create reusable sub-applications that can be loaded into different applications
- Integrate third-party applications into your main application

The way in which a sub-application is loaded defines the level of interoperability between the main application and the sub-application. Consider the following factors when loading a sub-application in your main application:

Trusted applications What level of trust do the applications have? A trusted sub-application has a greater amount of interoperability with the main application. An untrusted sub-application, while limited, can still interoperate with the main application in some ways. In general, though, if you do not have complete control over the development and deployment of a loaded sub-application, consider that application to be untrusted.

Versioning Are the main application and sub-application compiled with the same version of the Flex framework? The default method of loading a sub-application assumes that all applications are compiled by the same version of the framework. However, Flex can load a sub-application that was compiled with an older version of the framework. This type of application is known as a *multi-versioned* application. A multi-versioned application has some restrictions on its level of interoperability with the main application that loads it. It is, however, more flexible to use in a large application.

The level of trust and use of versioning are determined by the application domain and the security domain into which a sub-application is loaded.

There are three main types of loaded sub-applications in Flex:

Single-versioned applications are guaranteed to have been compiled with the same version of the compiler as the main application. They have the greatest level of interoperability with the main application, but they also require that you have complete control over the source of the sub-applications.

Multi-versioned applications can be compiled with older versions of the Flex framework than the main application that loads them. Their interoperability with the main application and other sub-applications is more limited than single-versioned applications.

Sandboxed applications are loaded into their own security domains, and can be multi-versioned. Using sandboxed applications is the recommended practice for loading third-party applications. In addition, if your sub-applications use RPC or DataServices-related functionality, you should load them as sandboxed.

When compiling each of these types of applications, you should include the MarshallingSupport class into the main application and sub-applications. You do this with the `includes` compiler argument, as the following example shows:

```
-includes=mx.managers.systemClasses.MarshallingSupport
```

Using sub-applications is in some ways like using modules. For a comparison of the two approaches, see "Comparing loaded applications to modules" on page 187.

More Help topics

"Developing sandboxed applications" on page 205

"Developing multi-versioned applications" on page 213

"Creating and loading sub-applications" on page 191

About application domains

An application domain is a container for class definitions. Applications have a single, top-level application domain called the system domain. Application domains are then defined as child nodes of the system domain. When you load a sub-application into another, main application, you can load it into one of three application domains: sibling, child, and current. When you load a sub-application into a *sibling* application domain, the sub-application's application domain has the same parent as the main application's application domain. In addition, it is a peer of all other sibling applications.

When you load a sub-application into a *child* application domain, the sub-application's application domain is a child of the main application's application domain. When you load a sub-application into a *current* application domain, the sub-application is loaded into the same application domain as the main application. Each of these locations defines where the sub-application can get its class definitions from.

The default behavior of the SWFLoader and Loader controls is to load a sub-application into a *child* application domain. If the sub-application and the main application are compiled with different versions of the Flex framework, runtime errors can result. These errors occur because the applications are sometimes compiled against different definitions of the same classes.

You can specify that the main application loads a multi-versioned sub-application. It does this by loading the sub-application into a *sibling* application domain. This means that the sub-application defines its own class definitions and does not get them from its parent. It is possible for two applications to work together, even if they are compiled with different versions of the Flex framework.

***Note:** In a multi-versioned application, the sub-applications must be compiled with the same or older version of the compiler that the main application was compiled with.*

You specify the application domain of a sub-application by setting the value of the `loadForCompatibility` property on the SWFLoader. If you set the value of this property to `true`, then the sub-application is loaded into a sibling application domain. If you set the value of this property to `false`, then the sub-application is loaded into a child application domain. The default value of this property is `false`, so by default, sub-applications are not multi-versioned.

You can also specify the application domain on the LoaderContext object. You do this if you specify the value of the `loaderContext` property when using the SWFLoader control. For more information, see "Specifying a LoaderContext" on page 195.

The system domain

Classes defined by Flash Player are in the system domain. The system domain parents all other application domains. The main application's application domain is a child of the system domain. If you load sub-applications into sibling application domains, then they are also children of the system domain. Classes defined in the system domain are never redefined in sub-applications or main applications. Those applications all share the common definitions of Flash Player. These definitions include classes such as DisplayObject, Event, and Sprite. The definitions of these shared classes are contained in the playerglobal.swc file.

Sibling application domains

The application domain that a sub-application is in determines where the sub-application gets its class definitions from. If the main application loads a sub-application into a sibling application domain, the sub-application defines its own non-player class definitions. This is the configuration for multi-versioned applications. Applications that are loaded into a sibling application domain can load other applications into a sibling application domain.

sub-applications that are in sibling application domains of the main application can communicate with the main application. Sub-applications can call methods and access properties on the applications, as long as they meet these criteria:

- Only strong types that are defined in Flash Player are used.
- The sub-applications are not in different security domains.

The ability for a sub-application in a sibling application domain to communicate with the main application is not as seamless as when the sub-application is in a child application domain. For example, if the sub-application launches a pop-up control, it passes an event to the main application that actually launches the control. This event passing limits their communication somewhat, but does provide some measure of interoperability. For more information about developing this type of application, see "Developing multi-versioned applications" on page 213.

Applications that are in separate security domains are automatically in separate, sibling application domains. As a result, if you load an untrusted sub-application into a main application, that sub-application has its own class definitions.

More Help topics

"Developing multi-versioned applications" on page 213

Child application domains

If a main application loads a sub-application into a child application domain of its application domain, the sub-application gets its class definitions from the main application. This behavior is the default for application loading. It can result in runtime errors if the applications are compiled with different versions of the Flex framework. As the SWF file sets up its classes, classes that are already defined are not added to the application domain. First in wins, which means that the first class defined becomes the only definition of that class. Subsequent definitions loaded into that application domain are ignored.

Sub-applications that are in child application domains of the main application can communicate with the main application. They have the highest level of possible interoperability with the main application. This situation is typical of a large application that is not multi-versioned, and it is the default behavior for the SWFLoader.

More Help topics

"Creating and loading sub-applications" on page 191

The current application domain

If you load a sub-application into the current application domain (rather than a separate, sibling application domain or a child application domain), the sub-application's class definitions are often ignored. This behavior is because the first definition in a domain is used. Subsequent definitions loaded into that domain are ignored. If new class definitions are added, the main application can use them.

Using the current application domain is typical of RSLs and other specially compiled resources, and is not typically used when loading sub-applications.

About security domains

Security domains define the level of trust between applications. The greater the trust between applications, the greater the amount of possible interoperability between those applications. In general, if a sub-application is loaded into the same security domain as the main application, then the applications have the highest level of interoperability.

If a sub-application is loaded into a different security domain (as is the case with many remote or multi-versioned applications), then the sub-application is allowed a limited amount of interaction with the main application. Sub-applications in separate security domains are also restricted in their ability to communicate with one another. They are known as sandboxed applications.

You determine whether a sub-application is loaded into the same security domain as the main application when you load it. You can set the value of the `trustContent` property to `true` to load a remote sub-application into the same security domain as the main application. This behavior only applies if that application is loaded from a different web domain or subdomain than the main application. If the sub-application is loaded from the same web domain as the main application, then it is by default loaded into the same security domain. Setting the value of the `loadForCompatibility` property does not affect the `trustContent` property.

In some cases, you want to load a sub-application that is on the same domain as the main application into a separate security domain. This might be because the sub-application is from a third party or you want you applications to have the same level of interoperability as a sandboxed application. One way to do this is to set up a different sub domain name on the same server, and load the sub-application from that sub domain. For more information, see "Loading same-domain and cross-domain applications" on page 189.

When using AIR, you cannot set the value of the `trustContent` property to `true`.

If you do not set the value of the `trustContent` property to `true`, then a sub-application on a remote server is loaded into a separate security domain by default.

You can also specify the security domain on the LoaderContext object. You do this if you specify the value of the `loaderContext` property when using the SWFLoader control. You can only do this if you want to load the sub-application into the same security domain. For more information, see "Specifying a LoaderContext" on page 195.

Sandboxed applications have the greatest number of limitations on application interoperability. These restrictions include the following:

Stage Access to the stage from the sub-application is limited to some stage properties and methods.

Mouse You cannot receive mouse events from objects in other security domains.

Pixels Applications cannot access the pixels drawn in applications that are in other security domains.

Properties While applications can get references to objects in other security domains, avoid doing this for security reasons. Some properties are restricted, such as the Stage or any parent of a DisplayObject that another application instantiates. In addition to these restrictions, applications that are in separate security domains are also in separate application domains by definition. As a result, they are subject to all restrictions of that architecture. However, they can also benefit from this situation because they can then be multi-versioned.

For more information about working with sandboxed applications, see "Developing sandboxed applications" on page 205

Common types of applications that load sub-applications

Common types of applications that load sub-applications include:

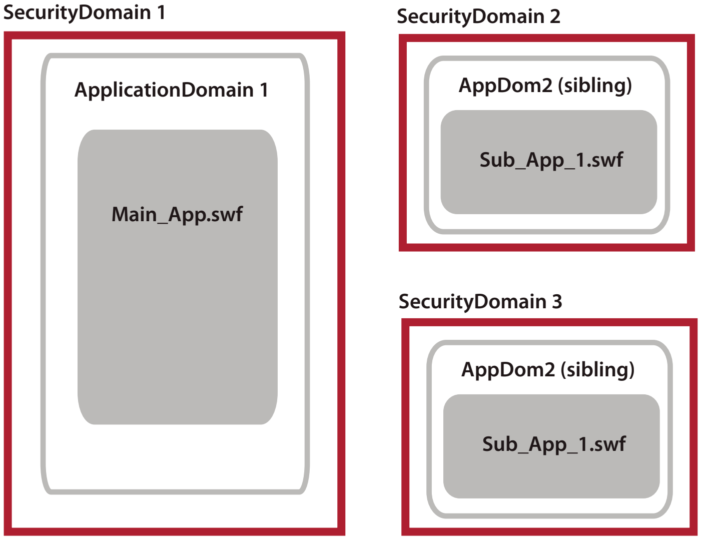
- Large applications that are or are not multi-versioned
- Mashups
- Portals
- Dashboards

About sandboxed applications

A sandboxed application is a common type of application that loads sub-applications. In general, sandbox applications load applications that are compiled and hosted by third parties. The main application does not necessarily trust the third party that developed those sub-applications. In addition, the developer of the main application does not know the version of the Flex framework that was used to compile the sub-applications (it must have been compiled with an older version or the same version of the compiler). As a result, sandboxed applications have the least amount of interoperability between the main application and the sub-applications, but they are typically multi-versioned. A common type of sandboxed application is a portal.

Sandboxed applications require the least amount of additional coding when using RPC classes and DataServices-related functionality. Trusted multi-versioned applications often require a bootstrap loader so that RPC classes work across applications. Sandboxed applications do not have this requirement. As a result, with many multi-versioned applications, using sandboxed applications is actually the recommended approach.

The following image shows the boundaries of a sandboxed application. The sub-applications are loaded into separate, sibling application domains. This means that each application contains its own class definitions so they can interoperate with applications that were compiled with different versions of the framework. The sub-applications are also in separate security domains. This separation means that they do not trust each other, and therefore have limited interoperability.



More Help topics

"Developing sandboxed applications" on page 205

About large, multi-versioned applications

A large, multi-versioned application is typically made up of many different components that are developed by different groups within an organization. For example, a form manager.

Multi-versioned applications load the sub-applications into a sibling application domain that is alongside the main application's application domain. They also load sub-applications into the same security domain as the main application.

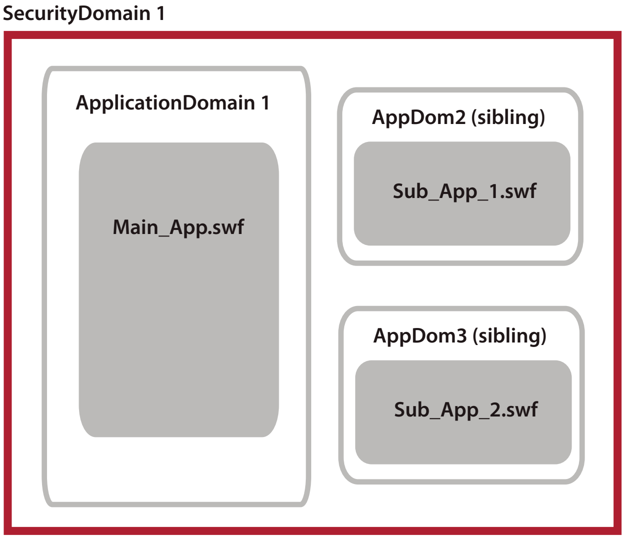
The sub-applications loaded by the main application in a multi-versioned architecture must have compiled with the same or older version of the compiler. You cannot load a sub-application into a main application if the sub-application was compiled with a newer version of the compiler.

Multi-versioned applications have many of the benefits of sandboxed applications, but can require additional coding to work with RPC classes and DataServices-related functionality. In general, if your sub-applications use these classes, use the sandboxed application approach for loading sub-applications, even if they are trusted. For more information, see "Using RPC and DataServices classes with multi-versioned applications" on page 214.

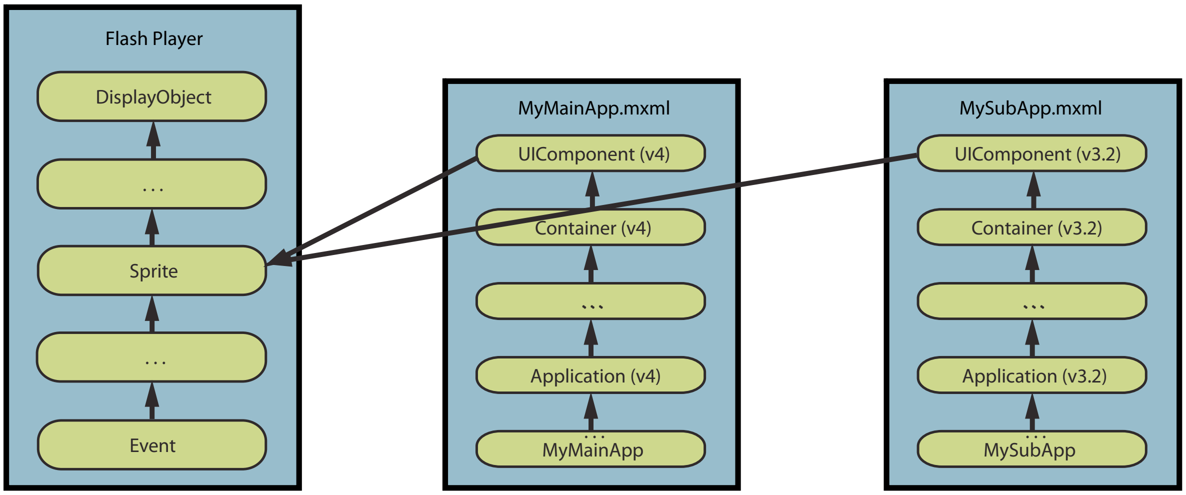
In deployment, the main application and the sub-applications are typically all in the same web domain, and so they have a trusted relationship. Even if they are deployed on different web domains, the applications are typically loaded into the same security domain to maintain the trusted relationship. This process of loading cross-domain applications into the same security domain is known as *import loading*. Use this process only under rare circumstances where you know that you can trust the source of the loaded sub-applications.

Sometimes third parties compile the sub-applications that the main application loads. In this case, developers sometimes don't know the version of the Flex framework that was used to compile the sub-applications. The sub-applications must be compiled with older versions of the compiler, or the same version of the compiler, but the developers of the main application do not need to know which version. They might not have access to the source code of the sub-applications. As a result, they would not necessarily be able to recompile them with the same version of the framework as the main application.

The following image shows the boundaries of a large, multi-versioned application. The sub-applications are loaded into separate, sibling application domains. This means that each application contains its own class definitions so they can be compiled with different versions of the framework (as long as the sub-applications are compiled with the same or older versions of the compiler that the main application was compiled with). All the applications are loaded into the same security domain and therefore trust each other.



Sub-applications in a sibling application domain store their own class definitions, apart from the class definitions in the main application. The following image shows the class definitions of a large, multi-versioned application. You can see that the sub-application uses its own definitions, which were compiled with Flex 3.2). The main application uses its own definitions, which were compiled with Flex 4.



More Help topics

"Developing multi-versioned applications" on page 213

About large, single-versioned applications

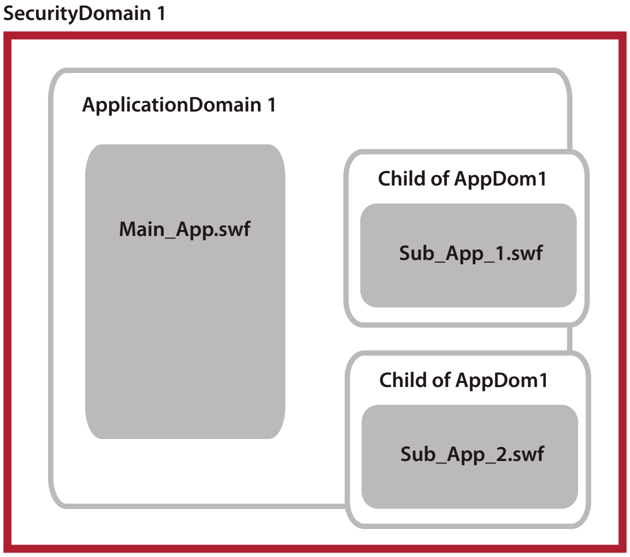
A large application without versioning support is an application that loads sub-applications that must have been compiled with the same version of the Flex framework. A single group within an organization typically creates these applications. It is possible for that group to enforce Flex framework versions and other standards during the development process.

Large, single-versioned applications are the default type of application loaded by the SWFLoader control. If you accept the SWFLoader's default settings when you load sub-applications into your main application, the sub-applications are not multi-versioned. They are loaded into child application domains of the main application.

It can be difficult to maintain large, single-versioned applications. This is because all sub-applications must be recompiled whenever any of the applications are recompiled with a new version of the framework. In addition, it is more difficult to add third-party sub-applications because they might have been compiled with different versions of the framework. In many cases, you do not have access to the source code to recompile them.

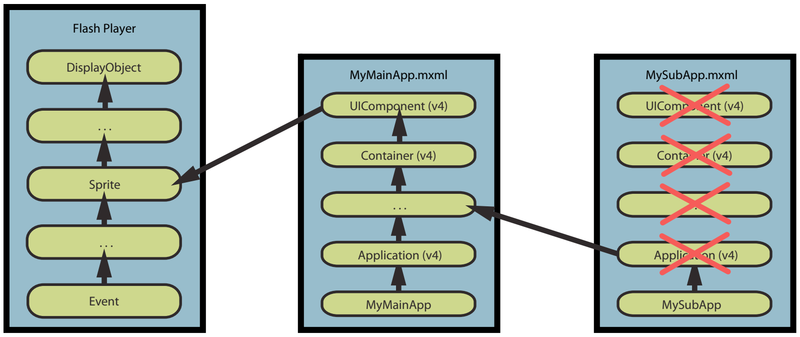
Large, single-versioned applications and all their sub-applications are usually deployed in the same web domain. This is because the same group within an organization typically develops and maintains them. As a result, they have a trusted relationship. Even if the applications are deployed on different web domains, the applications would be import loaded into the same security domain. This results in the applications having a trusted relationship.

The following image shows the boundaries of a large application that does not support multi-versioning. Each sub-application is loaded into a child application domain of the main application's application domain. As a result, all the applications use the same class definitions. If one of these applications is compiled with a different version of the framework, then runtime errors are likely to occur when a call is made to an API that is different. All the applications are inside the same security domain and therefore trust each other.



Sub-applications in a child application domain inherit their class definitions from the application in the parent application domain. If a sub-application defines one of the classes that is already defined in the main application, the child's definition is ignored. If multiple sub-applications define the same class that isn't defined in the main application, each sub-application uses its own definition.

The following image shows the class definitions of a large, single-versioned application. In this image, you can see that the sub-application gets its definitions from the main application, which was compiled with Flex 4. The class definitions in the sub-application, which was also compiled with Flex 4, are ignored.



More Help topics

"Creating and loading sub-applications" on page 191

About the manager classes

The manager classes handle various tasks of the application. For example, the DragManager handles all the drag-and-drop functionality; this manager class is responsible for marshaling data, creating the DragProxy, and triggering drag-related events.

When sub-applications are loaded into the same application domain as the main application, the application domain contains only a single instance of each manager. When applications are in different application domains, though, there can be more than one instance of a manager in the system domain.

Depending on the type of task, the manager classes are sometimes allowed to communicate through event passing when a main application loads a sub-application. For example, the FocusManager in a sub-application receives control from the FocusManager in the main application when the focus shifts to the sub-application. When the focus shifts away from the sub-application, the sub-application's FocusManager passes control back to the main application's FocusManager.

In other cases, there can only be one active instance of a manager in the system domain. Flex ensures that the sub-application's manager is disabled. For example, the BrowserManager cannot have multiple instances. Only the main application can access it, and only the main application can communicate with it, unless the sub-application is in a child application domain of the main application. If a sub-application in a separate application domain wants to communicate with the BrowserManager, it must do so through the main application.

In this case you would have to create custom logic that handles interaction with manager classes. For example, if you want a sub-application to use the main application's deep linking, you can create a custom class. This custom class passes messages from the sub-application to the main application, where the BrowserManager can be communicated with.

What typically happens is that the top-level manager handles the user interaction. This manager receives messages from the sub-application's manager that instructs it on what to do.

The following manager classes are linked in by all applications:

- SystemManager
- LayoutManager
- StyleManager
- EffectManager

- ResourceManager (which links in ModuleManager)
- FocusManager
- CursorManager
- ToolTipManager

The following manager classes are linked in only when used in an application:

- DragManager
- BrowserManager
- HistoryManager (deprecated)
- PopUpManager (note that if a sub-application uses pop-up controls, the main application must include a definition of this manager)

If a main application loads sub-applications that are compiled with the same version of the framework, only one definition of each manager is stored in the SWF file. However, if you have a sandboxed or multi-versioned sub-application, both the main application and the sub-application store their own definitions. In many cases, the manager classes in the sub-application pass messages to the main application's instance of the manager. The main application's manager can then handle the task.

With sandboxed or multi-versioned applications, both the main application and the sub-application have their own versions of the manager classes. In single-versioned applications, only one version of most manager classes is necessary. You can externalize the definition of a manager in many cases for applications that are not multi-versioned.

About the SystemManager class

All applications have an instance of the SystemManager class. If an application is loaded into another application, a SystemManager for the sub-application is still created. The SystemManager still manages the sub-application's application window, but it might not manage pop-up controls, tool tips, and cursors, depending on the way in which the sub-application was loaded. The content of the SWFLoader that loaded the sub-application contains a reference to the sub-application.

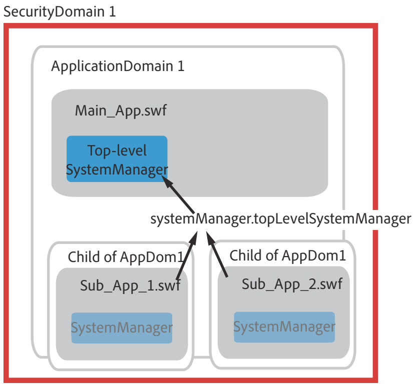
The SystemManager of the main application is important because it gives you access to many aspects of the entire application. For example, the top-level SystemManager does the following:

- Parents all pop-up controls, ToolTip objects, and cursors in the main application and all trusted sub-applications
- Handles focus for all applications that are trusted

You can use a reference to the top-level SystemManager to register to listen for events in a sub-application. For example, your sub-application can listen for mouse events that are outside its application domain by adding listeners to the top-level SystemManager.

The way you access the top-level SystemManager from a sub-application depends on the type of sub-application that you are using.

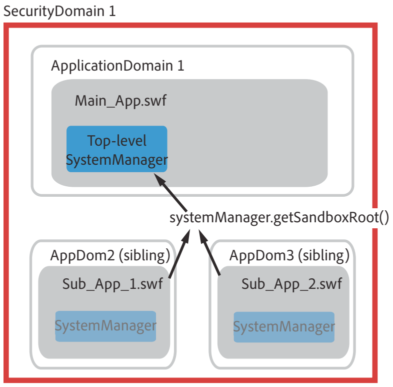
To get access to the top-level SystemManager in an architecture where sub-applications are loaded into child application domains, you use the `topLevelSystemManager` property of the SystemManager. The following image shows that the sub-applications use the `systemManager.topLevelSystemManager` to access the SystemManager in the main application.



For a code example that uses the `topLevelSystemManager` property to access the main application's SystemManager, see "Listening for mouse events with loaded applications" on page 201.

For single-versioned applications, the `topLevelSystemManager` property always refers to the top-level SystemManager in an application domain. If your applications are in separate application domains (as sandboxed or multi-versioned applications are), use the `getSandboxRoot()` method to get the top-level SystemManager for that security domain.

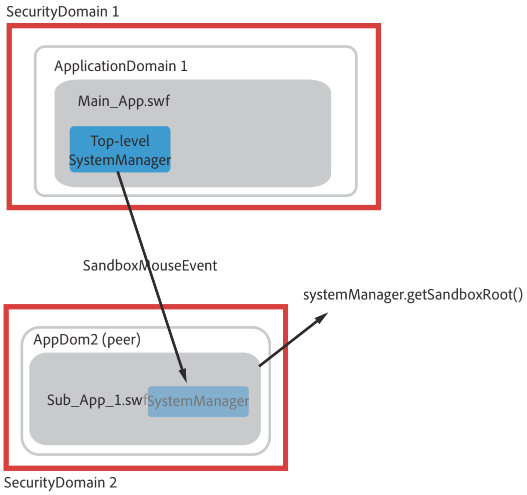
The following image shows that the sub-applications use the `systemManager.getSandboxRoot()` to get a reference to the main application's SystemManager, which is in a separate application domain.



For a code example that uses the `getSandboxRoot()` method to access the main application's SystemManager, see "Listening for mouse events in multi-versioned applications" on page 216.

When a sub-application is in a different security domain as the main application (or *not trusted*, such as in a sandboxed application), you cannot get a reference to the main application's SystemManager from the sub-application. In this case, you can listen for a SandboxMouseEvent and InterDragManagerEvent for inter-application communication. Any application can trigger these events. The sub-application's SystemManager gets notification of them, at which point you can handle them in the sub-application.

The following image shows an architecture where two applications, in separate security domains, communicate through event passing. The sub-application calls the `getSandboxRoot()` method to get a reference to its own SystemManager. The SystemManager can then listen for events that the main application's SystemManager dispatched.



The properties of a SandboxMouseEvent object are not the same as the properties in a typical event object. For example, the `target` and `currentTarget` properties are the SandboxBridge that dispatched the event, or the SystemManager that redispatched it. These properties are not the specific object that dispatched the event. Untrusted child applications should not be able to get references to objects that are typically specified by these properties. Similarly, the `stageX`, `stageY`, `localX`, and `localY` properties are not available.

For an example that uses the SandboxMouseEvent object to handle mouse events outside a sub-application's security domain, see "Listening for mouse events with sandboxed applications" on page 209.

Comparing loaded applications to modules

When you design a large application, give some consideration to its architecture. If the application has one main application that loads and unloads subordinate applications, then weigh the benefits of using either modules or sub-application when deciding which approach to take.

Consider the example of an application composed of many forms. In a modular application, a main application loads each form as a module with the ModuleManager. In an application that uses sub-applications, each form is a separate application that the SWFLoader loads into the main application. These two approaches are similar in many ways, but also have many differences.

Many of the reasons to use modules also apply to sub-applications. Main applications usually load sub-applications rather than embed their functionality. The result is generally a smaller initial download size and shorter load time for the main application. In addition, this promotes better encapsulation of related functionality, which can make development and maintenance easier.

Modules and sub-applications have the following similarities:

- Are compiled SWF files
- Can be loaded and unloaded at run time
- Promote encapsulation of related functionality
- Allow for an asynchronous development environment

- Can be recompiled without having to recompile the main application
- Support debugging
- Can be loaded locally or remotely (with the appropriate permissions)
- Are cached by the browser
- Can be preloaded
- Support progress events for getting the status while loading
- Can dynamically access methods and properties of the main application

Modules and sub-applications are also different. The main difference is that modules typically share classes with their host application. This sharing of classes creates a dependency between the module and the main application. Sub-applications, on the other hand, typically have their own versions of the classes, so less dependency exists between them and the main application.

Other differences between modules and sub-applications include the following:

File size Modules and their host applications are often smaller in file size because you can externalize shared classes. You can't always externalize shared classes for sub-applications. In cases where the versions of the applications are different, each application must have its own set of classes.

Reuse Modules are more tightly bound to the host application. They use interfaces to communicate with the loading application. This means that as your application changes, if your application uses modules, you'll probably need to recompile all of them if you move the main application to a later version of Flex.

Versioning Modules do not support multi-versioning. The main application and all modules must be compiled by the same version of the Flex framework. In a mult-versioned application, the sub-applications can be compiled with different versions of the compiler, as long as the compiler is the same or older than the compiler used to compile the main application.

Manager classes Modules and their host applications typically share manager classes. The modules are in a child application domain, whereas sub-applications often have their own instance of the manager class so that they can be multi-versioned. An exception is the StyleManager class; modules each define their own instance of type IStyleManager2.

ActionScript-only Modules can be either MXML-based or ActionScript-only; applications and sub-applications are not typically pure ActionScript.

Application domains sub-applications can be loaded into sibling application domains or child application domains, whereas modules must be loaded into a child application domain of the host application.

Security domains Modules must be loaded into the same security domain as their host application. Sub-applications can be loaded into either the same security domain or a different security domain.

For more information about modules, see "Modular applications" on page 138.

Comparing the SWFLoader and Loader controls

You can use the SWFLoader and Loader classes to load sub-applications into a main application. In most circumstances, use the SWFLoader class. This class wraps the Loader class and provides additional functionality that makes it easier to use for loading sub-applications into main applications.

The SWFLoader control has the following features:

- Supports Flex styles and effects; the Loader class does not have any inherent support for styles and effects.
- Lets you monitor the progress of a load inherently (if you use the Loader class, you have to first get a reference to a LoaderInfo object).

- Is a UIComponent. As a result, the SWFLoader control participates in the display list and adds children to the display list without having to write additional code.
- Resizes and scales the contents automatically.
- Does not require that the SWF file be an instance of the Application class, it just checks if an Application exists, and handles sizing differently.
- Can be multi-versioned. The Loader class does not have built-in support for multi-versioning.

Communicating across domains

Whenever you have a main application and sub-applications that are in different application domains, the recommended way to communicate between the applications is through event passing. If something that happens in the sub-application affects the main application, the sub-application triggers an event. This event can be caught in the main application. The event defines the properties necessary to respond to the event. For example, suppose a user moves the mouse pointer over a control in a sub-application that has a ToolTip. The sub-application's ToolTipManager creates the ToolTip and sends an event that requests the main application's ToolTipManager to display it. This interaction works across application domains that are within the same security domain.

Communication across security domains use the LoaderInfo.sharedEvents dispatcher and custom events. This process is largely transparent to you; you only need to know which events to register with event listeners. The LoaderInfo.sharedEvents class dispatches events in a sub-application that should be handled by a manager in the main application. The event contains data that define it, and that data is marshaled across the security domain. It is important to note that custom event classes cannot be strongly typed across the security domain by the receiver. Only classes defined by Flash Player can be strongly typed.

In many cases, communicating across security domains is transparent to the developer. The Flex framework handles the details of event passing and data marshaling for you. For example, when using functionality managed by the FocusManager or PopUpManager, you do not generally write additional code to make application interoperability work. The underlying Flex classes trigger the events and marshal the data across the domains so that the experience is seamless. In some cases, such as when you use custom classes to define objects, you must write custom code to marshal the data across the security domain.

Loading same-domain and cross-domain applications

The level of interaction that a main application and sub-application have depends on the level of trust between them. They can be trusted or untrusted. By default, applications that are loaded from a different web domain are untrusted, and applications that are loaded from the same web domain are trusted. You can, however, make a cross-domain application trusted.

Same-domain applications are applications that are loaded into the main application from the same web domain as the main application. By default, they are loaded into the same security domain and are trusted. Trusted applications can be loaded into the same application domain as the main application, which means they share their class definitions with the main application. They can also be loaded into a different, sibling application domain, which means that they contain their own definitions for all their classes and can be multi-versioned (as long as they are compiled with the same or older version of the compiler that the main application was compiled with).

Cross-domain applications are applications that are loaded into a main application from a different web domain as the main application. For example, if the main application is located at domainA.com, and the sub-application is located at domainB.com, then the sub-application is a cross-domain application. Cross-domain applications are often known as untrusted applications, although you can specify that a remote application be trusted (except when using AIR).

When loading cross-domain SWF files, consider these points:

- The main application might have to call the `Security.allowDomain()` method.

- The sub-application's server might require a crossdomain.xml if you try to load data from the target application.
- The main application can define the level of trust between the main application and the sub-application by setting the value of the `trustContent` property (if the crossdomain.xml file permits it, and you are not using AIR).

Untrusted applications are loaded into a different security domain as the main application. As a result, Flash Player provides only limited interoperability between applications that do not trust each other because of the security concerns.

You can load the content of any SWF file from any web domain regardless of whether target web server has a policy file, but you need permissions to read the SWF file's data. To load a cross-domain application and access its data, the target server that hosts the remote sub-application must have a policy file called crossdomain.xml on it. This file must specifically allow access from the source server. In the previous example, if you wanted to access the loaded application's data, domainB must have a policy file on it that lets domainA load from it. Without a policy file, Flash Player throws a security error if it tries to access the application's data. A policy file must be in the root of the domain, or its location must be specified explicitly in the loading application. For more information about the crossdomain.xml file, see "Using cross-domain policy files" on page 125.

Sub-domains that are not the same, such as www1.domainA.com and www2.domainA.com, also apply when deciding if a sub-application is cross-domain or not. In this example, if the main application is on www1.domainA.com, and the sub-application is on www2.domainA.com, then the sub-application is considered untrusted by default. To load data from the sub-application, the sub-application's domain would require a policy file.

You cannot test loading cross-domain applications directly in Flash Builder because the sub-applications must be loaded from a separate domain. As a result, to build the main application and the sub-application in Flash Builder, set up the compilation process to deploy the applications to separate servers for testing.

Cross-domain applications can be "import loaded," which means that they are loaded into the same security domain as the main application. You do this by setting the `trustContent` flag to `true` on the SWFLoader, or by specifying that the application load into the same security domain domain on the LoaderContext. You should only import load a cross-domain application if you know for sure that it can be trusted. For more information on using the LoaderContext to load sub-applications, see "Specifying a LoaderContext" on page 195.

Another way to load cross-domain applications into the same security domain is to use the `Security.allowDomain()` method. You first load the application as a sandboxed application (from a different web domain, without setting the `trustContent` property to `true`). In the main application, you call the `allowDomain()` method on the sub-application's domain in the SWFLoader control's `complete` event handler. In the sub-application, you call the `allowDomain()` method on the main application's domain. You must make this call early in the sub-application's lifecycle. For example, in the application's `preinitialize` event. If you use the `allowDomain()` method to load applications, you do not need a crossdomain.xml file on the target domain.

You can load a local (or same-domain) application into a different security domain. To do this, you can use a URL string for the SWFLoader's `source` property that is different from the path to the main application. For example, you could specify an IP address for the `source` property. When Flash Player compares the domain names, they are recognized as different, and the sub-application loads as an untrusted application. This is a common approach if you want to the loaded application to be sandboxed, but deploy the applications into the same web domain. You could also create separate sub domains on the same server to ensure that the sub-application is loaded into a separate security domain.

Sub-applications access remote data based on the way that they were loaded. If you import load an application, the sub-application is effectively running from within the main application's security domain. As a result, it would now need permissions to access data on its domain of origin. If you load a sub-application into a separate security domain, it runs within its original domain and can access resources on that domain as normal.

Creating and loading sub-applications

You use the SWFLoader control to load sub-applications into your main application. The default behavior of the SWFLoader control assumes that the application you are developing loads trusted sub-applications that were compiled with the same version of the Flex framework. These applications are typically loaded from the same web domain as the main application.

You can load the following other types of applications:

Sandboxed applications Sandboxed applications contain sub-applications that are loaded into separate security domains. As a result, they can be multi-versioned, but are untrusted. This is the recommended approach for any third-party application, or for many multi-versioned applications that use RPC classes or DataServices-related functionality. For more information, see “Developing sandboxed applications” on page 205.

Multi-versioned applications Multi-versioned applications are typically very large applications that load trusted sub-applications. The sub-applications might or might not have been compiled with the same version of the Flex framework as the main application. They must be loaded by the same or older version of the compiler that is used for the main application. For more information, see “Developing multi-versioned applications” on page 213.

When compiling each of these types of applications, you should include the MarshallingSupport class into the main application and sub-applications. You do this with the `includes` compiler argument, as the following example shows:

```
-includes=mx.managers.systemClasses.MarshallingSupport
```

When developing large, single-versioned applications, you might consider using modules instead of sub-applications. For more information, see “Comparing loaded applications to modules” on page 187.

Sub-applications can run independently of the main application and other sub-applications. There should be no dependencies on the main application or other sub-applications for a sub-application to run.

When developing sub-applications in Flash Builder, you cannot just compile the main application and expect the sub-application to also recompile. You must compile them separately. Using Ant or some other automated build process can help. This process is different from developing modules, where if you compile the application that loads a module, Flash Builder also compiles the module.

Loading applications with the SWFLoader control

The SWFLoader control lets you load one application into another application. It has properties that let you scale its contents. It can also resize itself to fit the size of its contents. By default, content is scaled to fit the size of the SWFLoader control. The SWFLoader control can also load content on demand programmatically, and monitor the progress of a load operation.

To load a SWF file into a SWFLoader control, you set the value of the SWFLoader control’s `source` property. This property defines the location of the sub-application’s SWF file. After you set the value of the `source` property, Flex imports the specified SWF file and runs it. For simple examples of using the SWFLoader tag in an application, see “Creating a SWFLoader control” on page 716.

In addition to SWF files, the SWFLoader control can also load several types of images (such as JPG, PNG, and GIF files) as SWF files. You cannot load a module, RSL, or style module with the SWFLoader control.

The default behavior for the SWFLoader control depends on the location of the loaded SWF file. If the loaded application is loaded locally, the default is to load the SWF file into the same security domain, and a child application domain. This loads an application that is trusted but single-versioned.

If the loaded application is loaded from a different server or in a separate sub domain than the main application, then the default is to load the application into a separate security domain and a separate, sibling application domain. The result is a multi-versioned application, but one that has some restrictions on it because it is untrusted by default. This is the recommended approach for all third-party applications, or any multi-versioned applications that use RPC classes or DataServices-related functionality.

When loading any application, you can explicitly set the values of properties on the SWFLoader control that set the application domain and the security domain. These properties define the trust level between the main application and the sub-application, and determine whether the applications are multi-versioned. The following table describes these properties of the SWFLoader control.

Property	**Description**
`loadForCompatibility`	Determines if the loaded SWF file is a multi-versioned application. Set this property to `true` to indicate that the loaded application might be compiled with a different version of the Flex framework and you want your application to be able to interact with it. Setting the value of the `loadForCompatibility` property to `true` also causes the LoaderContext of the Loader to load the sub-application into a sibling application domain of the main application, rather than a child application domain. ***Note:** The sub-application must be compiled with the same or older version of the compiler than the version that the main application was compiled with.* Set this property to `false` to disallow any version compatibility. If the loaded application was compiled with an older version of the framework, then the application will likely experience errors at run time (if the two applications use a shared resource whose API has changed between versions of the Flex framework). If the application was compiled with the same version of the framework, then it will load and interact with the main application as normal. The value of the `loadForCompatibility` property is ignored if you explicitly set the value of the `loaderContext` property. Setting this property has no impact on the sub-application's security domain. The default value of the `loadForCompatibility` property is `false`. If you set `loadForCompatibility` to `true` when developing your applications, use this method of loading consistently in the future. If you change an application from one that is multi-versioned to one that is single-versioned, or vice versa, you should not expect the application to work the same way. For more information, see SWFLoader.loadForCompatibility.
`loaderContext`	Defines the context into which the child SWF file is loaded. The context defines the application domain and the security domain of the application. With the `loaderContext` property, you can set the security domain into which the application is loaded. You can also require that a policy file is in place before the SWF file is loaded. The default for a local application is to load the sub-application into the same security domain as the main application. For a remote (or cross-domain) application, the default is to load the sub-application into a different security domain. With the `loaderContext` property, you can also specify the application domain for the context. The default is a child of the Loader's application domain. If you want to load a multi-versioned application, change this to a sibling application domain. In general, use the `loadForCompatibility` property rather than the `loaderContext` property to specify the application domain. If you explicitly set the value of the `loaderContext` property, then the SWFLoader control ignores the value of its `loadForCompatibility` and `trustContent` properties. The default value of the `loaderContext` property is `null`. You can only specify a LoaderContext in ActionScript. There are some restrictions on using the `loaderContext` property. For more information, see "Specifying a LoaderContext" on page 195. For more information, see SWFLoader.loaderContext.

Property	Description
trustContent	Determines whether the SWF file is loaded into the main application's security domain. Set to true to load the SWF file into the main application's security domain. Regardless of whether the SWF file is same-domain or cross-domain, the SWF file is now considered trusted (assuming that the policy file permits it to be). This means that it can access properties and methods of the main application, and vice versa. You should not set this to true for any third-party application unless you can absolutely trust the source of that application. Set this property to false to load a cross-domain SWF file into a separate security domain. It is now considered untrusted. The SWF file cannot access properties and methods of the main application, nor can the main application access the SWF file's properties and methods. The value of the trustContent property is ignored if you explicitly set the value of the loaderContext property. The default value of the trustContent property is false. For more information, see SWFLoader.trustContent.

Specifying a LoaderContext

Specifying a loader context for the sub-application gives you control over the security domain and application domain that the application is loaded into. You specify the loader context by setting the values of the `securityDomain` and `applicationDomain` properties on the LoaderContext object.

You can only specify a loader context in ActionScript. You cannot set its value as an attribute in the `<mx:SWFLoader>` tag.

The following example assigns a custom LoaderContext to the SWFLoader. It defines the current security domain and a sibling application domain for the loader. This is only applicable to sub-applications that are single-versioned. For multi-versioned applications or sandboxed applications, you should use calls to the `allowDomain()` method to establish trust between the main application and the sub-application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainAppCustomLoaderContext.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp()">
     <fx:Script>
          <![CDATA[
               import flash.system.SecurityDomain;
               import flash.system.ApplicationDomain;

               private function initApp():void {
                    var context:LoaderContext = new LoaderContext();

                    /* Specify the current application's security domain. */
                    context.securityDomain = SecurityDomain.currentDomain;
                    /* Specify a new ApplicationDomain, which loads the sub-app into a
                      peer ApplicationDomain. */
                    context.applicationDomain = new ApplicationDomain();
                    contentLoader.loaderContext = context;
                    contentLoader.source = "http://yourdomain.com/SubApp3.swf";
               }
          ]]>
     </fx:Script>
     <mx:SWFLoader id="contentLoader"/>

</s:Application>
```

This example triggers the `initApp()` method on the Application object's `creationComplete` event. If you had instead set the `source` property of the SWFLoader control in MXML, you would use the `preinitialize` event to set the loader context. Waiting for the application's `creationComplete` event to set the loader context would occur too late in the application startup.

Setting the `context.securityDomain` property to the `currentDomain` makes the SWF file trusted by loading it into the same security domain as the main application. To do this, though, the sub-application's server must have a policy file that allows you to load it. The result is that the SWF file becomes trusted, and acts as if it was loaded locally. In AIR, you cannot specify any value for the `context.securityDomain` property.

You can only specify the value `SecurityDomain.currentDomain` for the `context.securityDomain` property. Attempting to pass any other security domain results in a SecurityError exception. If you do not specify any value for the `currentDomain` property, then a remote SWF file is loaded into its own security domain. This occurs unless you set the security domain in some other way, such as setting the value of the SWFLoader's `trustContent` property to `true`.

When specifying an application domain for the `context.applicationDomain` property, you can add a sub-application to a sibling application domain, a child application domain, or the same application domain as the loading application. The following table describes each of these methods.

Value of context.applicationDomain	Result
`new ApplicationDomain()`	Loads the sub-application into a sibling application domain that is rooted on the same domain as the loading application. If the loading application is the top-level application, then both the loading application and the sub-application's application domain will be children of the system domain. You do this if you want to use compatibility mode, because applications that are in sibling application domains can each have their own class definitions. As a result, they can be compiled by different versions of the Flex framework, as long as the sub-applications are compiled with the same or older version of the compiler than the main application.
`new ApplicationDomain( ApplicationDomain.currentDomain)`	Loads the sub-application into an application domain that is a child of the main application's application domain. All class conflicts are resolved when the application is loaded; the first class definition wins. Do not use this method if you want to load a multi-versioned application. That is because applications that are loaded into the same application domains must be compiled by the same version of the Flex framework.
`application.currentDomain`	Loads the sub-application into the same application domain as the main application. You typically do not use this setting when loading sub-applications. It is generally only used for RSLs and other specially compiled resources.

If you specify a loader context when loading a sub-application, do not load more than one application into the same application domain. In other words, do not use the same loader context for more than one sub-application.

Unloading applications with the SWFLoader control

To unload a sub-application that you loaded with the SWFLoader control, call the SWFLoader control's `unloadAndStop()` method, as the following example shows:

```
myLoader.unloadAndStop(true);
```

You can also set the SWFLoader's `source` property to `null`. This results in a call to the `SWFLoader.content.loaderInfo.loader.unload()` method. You can call this method explicitly, but only for trusted applications.

To free up the memory that was used by the loaded sub-application, ensure that no references to objects or classes in the sub-application exist. The `unload()` method frees the loader's reference to the sub-application's bytes, but if code in the sub-application is still in use, then it is not garbage collected.

Flash Player also unloads a sub-application when the same SWFLoader control loads a new sub-application. The original content is removed before the new application is loaded.

Typically, you load a style module into the main application's application domain. If the sub-application contains any user interface classes (such as skins) that are not in the main application, the styles for those classes are loaded into the main application's StyleManager. When you unload the sub-application, the sub-application's memory is not freed up. In this case, load style modules into the sub-application's application domain.

Accessing the main application from sub-applications

There are several ways to access the methods and properties of the main application from the sub-application. These ways only work for sub-applications that are loaded as children into a main application's application domain. You cannot use these for sandboxed applications, or for multi-versioned applications.

These methods include:

- Using the `application` property of the Application class. This property accesses the root application from anywhere in the application. For more information, see "Accessing document and application scopes" on page 404.

- Using the parentDocument property of the Application class. This is useful if you have multiple applications embedded and want to just access the immediate parent. For more information, see "Accessing document and application scopes" on page 404.

Be aware that you might encounter security errors if you try to access members that should not be accessed.

Accessing sub-applications from the main application

Although you can access the sub-application from the main application, you cannot directly reference methods and properties. This is because the members of the sub-application, unless it is embedded at compile-time, are not known by the compiler when the main application compiles.

You can, however, get a reference to the sub-application's SystemManager. You can then treat members of the sub-application as dynamic properties and methods of the sub-application's object.

The following example loads a remote sub-application into the SWFLoader. It sets the value of the trustContent property to true so that the sub-application is loaded into the same security domain as the main application. It casts the content property of the SWFLoader to a SystemManager so that the application's members can be accessed dynamically. It then calls a method and accesses a property of the sub-application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainAppUsingSubAppMembers.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <fx:Script>
          <![CDATA[
               import mx.managers.SystemManager;
               private function getValueFromSubApp():void {
                    /* Cast the SWFLoader's content as a SystemManager
                      to access the sub-application. */
                    var subApp:SubApp2 =
                         (contentLoader.content as SystemManager).application as SubApp2;
                    /* Call a method and access a property of the
                      sub-application. */
                    label1.text = subApp.doSomething() + subApp.answer;
               }
          ]]>
     </fx:Script>

     <mx:SWFLoader id="contentLoader" visible="false"
          height="0" width="0"
          trustContent="true"
          source="http://yourdomain.com/SubApp2.swf"/>
     <mx:Panel id="myPanel2"
          paddingLeft="10" paddingBottom="10"
          paddingTop="10" paddingRight="10">
          <s:Label id="label1"/>
          <s:Button id="b2" label="Call SubApp2" click="getValueFromSubApp()"/>
     </mx:Panel>

</s:Application>
```

The following example shows the sub-application that is loaded by the main application in the previous example. It defines a public property, `answer`, as well as a public method, `doSomething()`, that returns a String.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/SubApp2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
          <![CDATA[
               // Define a public property.
               public var answer:int = 42;

               // Define a public method that returns a String.
               public function doSomething():String {
                    return "The answer is ";
               }
          ]]>
     </fx:Script>
</s:Application>
```

For more information about accessing sub-applications from the main application, see "SWFLoader control" on page 715.

If the sub-application is loaded remotely, you can access its members only if it is in the same security domain as the main application, or if you and the sub-application call the `Security.allowDomain()` method. In this case, you must call the `Security.allowDomain()` method from the main application on the sub-application's domain, and the sub-application must call this method on the main application's domain. You must also call the `allowDomain()` method early in the sub-application's lifecycle. For example, call it in a `preinitialize` event handler.

Creating class instances from loaded applications

You can create instances in a main application of classes that are defined in a loaded sub-application. You can then add these objects to your main application and interact with them as you would interact with any other object in the display list. To access class definitions in a sub-application, you get the definitions from the sub-application's application domain.

Each application domain contains definitions of all the classes within it. There is an applicationDomain object that you access with a reference to the loaded application's LoaderContext. The ApplicationDomain object has two methods, `hasDefinition()` and `getDefiniton()`. The `hasDefinition()` method lets you detect if a class definition exists. If a class definition exists, the `getDefinition()` method lets you create an instance of that class in your main application.

You can't add a UIComponent that is defined in another application domain to the main application's display list. As a result, to create an instance of a class that is defined in another application domain, the sub-application must be loaded into a child application domain of the main application's application domain (you cannot set the value of the SWFLoader's `loadForCompatibility` property to `true`). The sub-application must also be in the same security domain as the main application (`trustContent` must be `true`).

You can only create the instance of the class dynamically when the class is defined in an application that is loaded at run time. This is because the compiler does not have access to classes in loaded applications at compile time, so it cannot check linkages. If the sub-application was embedded at compile time, then you would not have to create the instance dynamically, the class definition would be available to the compiler. In this case, you could instead use the `new` keyword with the specific class type rather than a generic Class type.

The following example main application loads a sub-application with the SWFLoader control. It sets the value of the `trustContent` property to `true` and defines a method, `createClassInstance()`. This method gets the definition of the custom class from its loaded application's application domain. The example then creates an instance of this class and sets a property on it before adding it to the display list.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainAppUsingSubAppDefinitions.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <fx:Script>
          <![CDATA[
               import mx.core.UIComponent;
               public function createClassInstance():void {
                    // Check that the definition exists.
                    if
(contentLoader.loaderContext.applicationDomain.hasDefinition('MyRedButton')) {
                         var objClass:Class =
contentLoader.loaderContext.applicationDomain.getDefinition('MyRedButton') as Class;
                         if (objClass != null) {
                              var newObject:UIComponent = UIComponent(new objClass());
                              // Set properties on the custom class as an associative array.
                              newObject["label"] = "Click Me";

                             // Add the new instance to the second panel in this application.
                              myPanel2.addChild(newObject);
                         }
                    }
               }
          ]]>
     </fx:Script>
     <!-- The SWFLoader in the first panel loads the
           sub-application that contains a definition of MyRedButton. -->
     <mx:Panel id="myPanel" title="SubApp1 Loaded by main application">
          <mx:SWFLoader id="contentLoader"
            trustContent="true"
            source="SubApp1.swf"/>
     </mx:Panel>
     <!-- This application adds an instance of the MyRedButton
           class to the second panel after the content is loaded. -->
     <mx:Panel id="myPanel2" title="Instance of a Class Defined By SubApp1"/>

</s:Application>
```

In this case, the `hasDefinition()` method returns a boolean to tell if a class is present in the loaded SWF file. If it is present, the sub-application can obtain the Class using the `getDefinition()` method. You can then create instances by using the `new` keyword with the returned class.

Also notice that the `createClassInstance()` method is public. If it were private, the sub-application would not be able to call it.

The sub-application, SubApp1.swf, statically links a custom class called MyRedButton. It also calls the parent application's method when it is done. You cannot call that method in the parent application's `applicationComplete` event, because that event will likely be triggered too early in the initialization process. Wait for the sub-application to complete its loading and initialization before you create an instance of a class that is defined in it.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/SubApp1.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:custom="*"
    creationComplete="initApp()"
    applicationComplete="mx.core.FlexGlobals.topLevelApplication.createClassInstance()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

     <fx:Script>
          <![CDATA[
               private function initApp():void {
                    var child:DisplayObject = getChildAt(0);
                    var childClassName:String = getQualifiedClassName(child);

                   // Show that the qualified class name of the custom button is MyRedButton.
                    trace(childClassName);
               }
          ]]>
     </fx:Script>
     <custom:MyRedButton id="myRedButtonId" label="Click Me"/>
</s:Application>
```

The MyRedButton class is a simple MXML component that extends Button and defines its color as red. The following example shows this custom class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MyRedButton.mxml -->
<s:Button
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    color="red">
</s:Button>
```

Listening for mouse events with loaded applications

It is possible to listen for mouse events in a main application from a loaded sub-application that is in a child application domain. To listen for these events, you listen for events such as `MOUSE_UP` and `MOUSE_MOVE` on the sub-application's `systemManager.topLevelSystemManager`. When the sub-application is in a child application domain, the `topLevelSystemManager` property refers to the main application's SystemManager.

The following application shows how to access mouse events on the `topLevelSystemManager` in a sub-application that was loaded into a child application domain. If the sub-application is not in a child application domain, then you must handle access to the SystemManagers differently. For more information, see the examples in "Listening for mouse events with sandboxed applications" on page 209 and "Listening for mouse events in multi-versioned applications" on page 216.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/ZoomerPattern2.mxml -->
<s:Application
    creationComplete="setup()"
    height="250"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <fx:Script>
     <![CDATA[
          import mx.core.UIComponent;
          import mx.managers.PopUpManager;
          [Bindable]
          public var data1:Array = ["Ice Cream", "Fudge", "Whipped Cream", "Nuts"];
          public var zoomTool:UIComponent;
          public function setup():void {
               // Draw the zoom rectangle.
               zoomWidget.graphics.lineStyle(1);
               zoomWidget.graphics.beginFill(0, 0);
               zoomWidget.graphics.drawRect(0, 0, 17, 17);
               zoomWidget.graphics.endFill();
               // Listen for mouse down events.
               zoomWidget.addEventListener(MouseEvent.MOUSE_DOWN, zoom_mouseDownHandler);
          }
          private var lastX:int;
          private var lastY:int;
          private function zoom_mouseDownHandler(event:MouseEvent):void {
               // When the mouse is down, listen for the move and up events.
               systemManager.topLevelSystemManager.addEventListener(
                    MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
               systemManager.topLevelSystemManager.addEventListener(
                    MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
               // Update the last position of the mouse.
               lastX = event.stageX;
               lastY = event.stageY;
               // Create and pop up the zoomTool. This is what is dragged around.
               // It must be a popup so that it can float over other content.
               zoomTool = new UIComponent();
               PopUpManager.addPopUp(zoomTool, this);
               var pt:Point = new Point(zoomWidget.transform.pixelBounds.x,
                    zoomWidget.transform.pixelBounds.y);
               pt = zoomTool.parent.globalToLocal(pt);
               zoomTool.x = pt.x;
               zoomTool.y = pt.y;
               zoomTool.graphics.lineStyle(1);
               zoomTool.graphics.beginFill(0, 0);
               zoomTool.graphics.drawRect(0, 0, 17, 17);
               zoomTool.graphics.endFill();
               // Hide the rectangle that was the target.
               zoomWidget.visible = false;
          }
          private function zoom_mouseMoveHandler(event:MouseEvent):void {
               // Update the position of the dragged rectangle.
               zoomTool.x += event.stageX - lastX;
```

```
                zoomTool.y += event.stageY - lastY;
                lastX = event.stageX;
                lastY = event.stageY;
                var bm:BitmapData = new BitmapData(16, 16);

                // Capture the bits on the screen.
                bm.draw(DisplayObject(systemManager.topLevelSystemManager), new
                     Matrix(1, 0, 0, 1,  -zoomTool.transform.pixelBounds.x - 2,
                     -zoomTool.transform.pixelBounds.y - 2));
                // Create a Bitmap to hold the bits.
                if (zoomed.numChildren == 0) {
                     var bmp:Bitmap = new Bitmap();
                     zoomed.addChild(bmp);
                } else
                     bmp = zoomed.getChildAt(0) as Bitmap;
                // Set the bits.
                bmp.bitmapData = bm;
                // Zoom in on the bits.
                bmp.scaleX = bmp.scaleY = 8;
           }
           private function zoom_mouseUpHandler(event:Event):void {
                // Remove the listeners.
                systemManager.topLevelSystemManager.removeEventListener(
                     MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
                systemManager.topLevelSystemManager.removeEventListener(
                     MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
                // Replace the target rectangle.
                zoomWidget.visible = true;
                // Remove the dragged rectangle.
                PopUpManager.removePopUp(zoomTool);
           }
     ]]>
     </fx:Script>

     <mx:HBox>
          <mx:HBox backgroundColor="0x00eeee" height="140" paddingTop="4" paddingRight="4">
               <mx:Label text="Drag Rectangle"/>
               <mx:UIComponent id="zoomWidget" width="17" height="17"/>
               <mx:Canvas id="zoom"
                    borderStyle="solid"
                    width="132"
                    height="132"
               >
                    <mx:UIComponent id="zoomed" width="128" height="128"/>
               </mx:Canvas>
          </mx:HBox>
          <mx:List dataProvider="{data1}"/>
     </mx:HBox>
</s:Application>
```

The following example main application loads the previous example application. It uses the default settings to load the sub-application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainZoomerPattern2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <mx:Text text="Default (trusted application in child ApplicationDomain):"/>

     <mx:SWFLoader id="swf1" source="ZoomerPattern2.swf"/>

</s:Application>
```

In this example, you cannot drag the rectangle outside the sub-application's boundaries, and the `mouseUp` event does not get triggered.

This example application uses the `systemManager.topLevelSystemManager` property to get a reference to the main application's SystemManager. If the application were a stand-alone application, you could use the `systemManager` property to register event listeners, as the following example shows:

```
systemManager.addEventListener(MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
systemManager.addEventListener(MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
```

This works when the application is a stand-alone application, but not when the application is loaded as a child. The top-level SystemManager, which is not the sub-application's SystemManager, parents the zoomTool class.

Embedded fonts in loaded applications

Applications and modules in the same application domain are able to use the same embedded fonts by specifying the font name. For example, a font embedded in the main application can be used by a sub-application, as long as the applications are in the same application domain.

Models and singletons in loaded applications

Models that are implemented as singletons, and other singletons that the main application and its sub-applications share, do not work if the sub-application and main application are in separate application domains.

To share models and other singletons, you can write your own marshaling code or create a bootstrap loader that stores definitions of those classes.

Adobe recommends not sharing data models and other singletons with untrusted applications.

More Help topics

"Bootstrap loading" on page 219

Using RSLs with sub-applications

Single-versioned applications can share RSLs with other sub-applications and the main application to minimize the number of classes that are loaded. This results in faster load times and smaller file sizes. Multi-versioned or sandboxed applications cannot share RSLs.

Starting with Flex 4.5, sub-applications work much more efficiently with RSLs. The main application only loads those framework RSLs that are needed at startup, and creates placeholders for all remaining framework RSLs. The sub-application does not try to load RSLs that are already loaded by the main application. If the sub-application needs one of the framework RSLs that are not initially loaded by the main application, then the sub-application loads the RSL.

In addition, when a sub-application loads an RSL, you can specify which domain the RSL is loaded into with the `application-domain` compiler argument. This applies to both framework RSLs and custom RLSs. This lets you load an RSL into the parent, current, or top-level application domains.

When you develop sandboxed or multi-versioned applications, you cannot always externalize overlapping class definitions, because the class definitions might be different. For example, if two applications are compiled with different versions of the Flex framework, then each application must have its own definition of the manager classes (such as the LayoutManager or the CursorManager) and other classes in the framework. As a result, do not externalize those classes by compiling the applications against the same RSLs.

In previous versions of Flex, you were required to initialize certain manager classes such as the FocusManager if you loaded a module or sub-application into a main application that did not use those manager classes. You are no longer required to do this if you load the sub-application into a sibling application domain where each application has its own class definitions. However, each security domain must include a definition of the PopUpManager class in case an untrusted sub-application displays a modal dialog box.

For information on using RSLs with sub-applications, see “Using RSLs with modules and sub-applications” on page 258.

Developing sandboxed applications

Sandboxed applications contain sub-applications that are loaded into separate security domains. By definition, they are therefore loaded into separate application domains as well. As a result, they can be multi-versioned, but are untrusted. Because they are untrusted, their interoperability with the main applications is limited. Types of sandboxed applications include portals, mashups, and dashboards.

Sandboxed applications must be compiled with the same or older version of the compiler that the main application is compiled with.

When compiling sandboxed applications, you should include the MarshallingSupport class into the main application and sub-applications. You do this with the `includes` compiler argument, as the following example shows:

```
-includes=mx.managers.systemClasses.MarshallingSupport
```

If you are using any third-party applications, you should load them as sandboxed applications. In addition, if you are using multi-versioned applications that use RPC classes or DataServices-related functionality, you should also consider loading them as sandboxed applications. Otherwise, you might be required to provide additional code such as a bootstrap loader.

In a sandboxed configuration, loaded sub-applications are typically not on the same domain as the main application. This means that the applications might not necessarily trust the loaded applications by default. In addition, all sub-applications are not necessarily always visible at the same time, so the main sandboxed application must be able to load and unload sub-applications at any time.

In sandboxed applications, each sub-application is loaded into a separate application domain and a separate security domain. The interoperability across security domains is very limited. The sub-application cannot access most stage properties, methods, and events. It cannot get mouse and keyboard events from other security domains. It also cannot perform drag and drop operations to or from the main application, and pop-up controls are clipped at the boundaries of the sub-application. Data sharing between the main application and sub-application requires marshaling.

If you try to use the parent chain of an application object to access properties of the main application from a sub-application in a different security domain, you will encounter security errors at run-time. In addition, you cannot access the application through the SWFLoader.content object.

For details about the architecture of a sandboxed application, see “About sandboxed applications” on page 180.

Pop-up controls in sandboxed applications

Pop-up controls are parented by the application at the sandbox root of their security domain. This is because the sandbox root handles requests to display a modal window from its children.

Because pop-up controls are parented by the sandbox root, centering a popup in a sandboxed application centers it in the area of the screen occupied by the sub-application and not the entire application. It also means that pop-up controls are sometimes clipped by scroll bars and masks on the sub-application.

A sub-application in a separate security domain from the main application has the following behavior:

- Launching a modal dialog box dims the entire application, but the pop-up can only be dragged within the boundaries of the sub-application.
- Centering a pop-up centers it over the sub-application, not the main application.
- Dragging pop-up controls works over the sub-application only. If you drag a pop-up outside the sub-application, it is clipped.
- Focus shifts to the pop-up control when you first launch a pop-up.

A sandboxed application cannot display a window or dialog box outside the bounds of its application. This rule prevents an untrusted application from phishing for passwords by displaying a dialog box on top of all the applications. When displaying a popup window, the PopUpManager checks if the parent application trusts it and if it trusts the parent application before asking the parent to host the window. If the parent hosts a window it is displayed over the parent's content as well as the child's content. If no mutual trust exists between a main and sub-application, then the PopUpManager hosts the dialog box locally so that it can only be displayed over the content of the application itself. But if the parent trusts the child, the dialog box is not clipped by the boundaries of the child’s application.

When a main application does not trust a sub-application, the main application’s SWFLoader uses masking with a scrollRect and scroll bars to keep the sandboxed application’s content restricted to its own application space.

Pop-up-related controls such as ColorPicker, ComboBox, DateField, PopUpButton, PopUpMenuButton, and Menu sometimes display their contents in unexpected ways if their normal position would cause them to be clipped.

Alert controls in sandboxed applications

Alerts, like other pop-up controls in sandboxed applications, are clipped at the edge of the loaded application. When an Alert is being displayed, the main application and all sub-applications are covered with a modal dialog box to prevent interaction with their controls. The blur effect only applies to the sub-application that launched the Alert box, and its child applications. The blur effect is not applied to the parent application or sibling applications.

Styles and style modules in sandboxed applications

The StyleManager does not pass styles from a parent application to a child in a different application domain or security domain. Similarly, a main application does not inherit styles from a sub-application. Therefore, either define styles within your sub-application and do not depend on the sub-application inheriting styles from the main application, or load a style module into the main application’s application domain.

Sub-applications each have their own StyleManager. When you load a style module in a sub-application, the classes themselves should be loaded into the child ApplicationDomain. The styles are loaded into the sub-application’s StyleManager.

If you want a main application and a sub-application to use the same runtime style sheets, load the style module into the main application’s application domain. Sub-application’s styles are merged with the main application’s styles. Main applications do not inherit styles that are defined in style modules that are loaded into sub-applications.

A style module must be compiled with the same version of the Flex framework as the application into which it is loaded. The main application and sub-application might not be able to load the same style module, unless they are compiled with the same version of the framework.

When loading a style module into a sub-application, if you don't specify an application domain, the module is loaded into a *sibling* application domain of the sub-application. This can result in an error when the sub-application tries to use classes that are defined in the style module.

To load a style module into a sub-application, load the style module into a *child* application domain of the sub-application. The `loadStyleDeclarations()` method has two optional parameters, `applicationDomain` and `securityDomain`. You use these properties to control the application domain and the security domain into which style modules get loaded.

The following example loads a style module into a child application domain of the sub-application:

```
private function loadStyle():void {
    /* Load style module into a child ApplicationDomain by specifying
       ApplicationDomain.currentDomain. */
    var eventDispatcher:IEventDispatcher = styleManager.loadStyleDeclarations(
        currentTheme + ".swf", true, false, ApplicationDomain.currentDomain);
    eventDispatcher.addEventListener(StyleEvent.COMPLETE, completeHandler);
}
```

For more information on using style modules, see "Loading style sheets at run time" on page 1547.

Fonts in sandboxed applications

Applications that are in the same application domain are able to use the same embedded fonts by specifying the font name. However, if the sub-application is loaded into a different application domain (as is the case with sandboxed applications), then the sub-application must embed the font to use it.

Focus in sandboxed applications

The FocusManager class in the main application and sub-application integrate to create a seamless focus scheme. Users can "tab through" the sub-application, regardless of whether the application is in the same or a different security domain as the main application. Shift tabbing also works. The FocusManager class is one of the few manager classes that supports interoperability even across security domains.

When an application is loaded, the main application keeps track of that SWF file in a list of focus candidates. When the user moves focus into the sub-application, the sub-application's FocusManager takes over focus duties until the user moves focus outside the sub-application. At that time, the main application's FocusManager resumes control.

When a pop-up is dismissed, the focus is moved to the last place that had focus. This behavior can be another pop-up or it can be in the main application.

When focus is on a control that is in a different security sandbox, calls to the `getFocus()` method on that application's FocusManager return `null`. Calls to the `UIComponent.getFocus()` method also return `null`.

The FocusManager's `moveFocus()` method lets you programmatically transfer focus to a control under the jurisdiction of another FocusManager. It also lets you transfer the control of focus to another FocusManager.

Focus management across application domains works even with modal dialog boxes. When a different top-level window is activated, the SystemManager deactivates the FocusManager in the formerly active top-level window. The SystemManager also activates the FocusManager in the other window and changes the depth level (z-order) of the windows.

Cursors in sandboxed applications

If the applications are in different security domains, then a custom cursor in the sub-application only appears over the area of the screen that is allocated to that sub-application. Moving the mouse outside the bounds of the sub-application passes control of the cursor to the main application's CursorManager.

These rules apply to the busy cursor as well. If a busy cursor is visible and you move the cursor to the main application that is in a different security domain, the cursor changes back to the last used cursor in the main application.

Localizing sandboxed applications

As with style modules, the main application cannot access resource modules used in a sub-application, and vice versa. Each sub-application must load its own resource modules.

Each multi-versioned application has its own ResourceManager instance. As a result, each sub-application has its own `localeChain`.

If more than one sub-application has resource bundles for the same locale with the same name, then the first one in wins. The contents of all resource bundles of that name in other sub-applications are ignored. Those sub-applications use the one that was defined first.

Like style modules, load resource modules into the child application domain of a sub-application. You control the application domain and the security domain into which resource bundles are loaded. The `loadResourceModule()` method of IResourceManager has two optional parameters, `applicationDomain` and `securityDomain`.

Also like style modules, all resource modules in an application must be compiled with the same version of the Flex framework. Do this whether that application is a main application or a sub-application. You cannot use multiple resource modules that were compiled with two different versions of the framework in the same main application or sub-application.

The following example loads a resource module into a child application domain of the sub-application:

```
private function loadBundle():void {
    /* Load resource module into a child ApplicationDomain by specifying
       ApplicationDomain.currentDomain. */
    var eventDispatcher:IEventDispatcher = ResourceManager.loadResourceModule(
        "MyBundle.swf", true, false, ApplicationDomain.currentDomain);
    eventDispatcher.addEventListener(StyleEvent.COMPLETE, completeHandler);
}
```

If you use run-time resource bundles with sub-applications, you should consider setting the `addResourceBundle()` method's `useWeakReferences` parameter to `true`. For more information, see "Preventing memory leaks in modules and sub-applications" on page 2118.

ToolTip objects in sandboxed applications

When in a different security domain, ToolTip objects are parented by the sub-application's SystemManager and are therefore clipped and masked by the main application. The ToolTipManager styles and positions the tip in a sub-application so that it fits within the sub-application's area of the screen only. If the ToolTip object is larger than the area of the sub-application, the ToolTip object is clipped.

ToolTip styles in the main application are not inherited by ToolTip objects in the sub-application.

This applies to error tips and data tips on List objects in sandboxed sub-applications as well.

Layouts in sandboxed applications

For controls that have pop-up or drop-down menus, when they are in a separate security domain, they are initially displayed unclipped. These controls are restricted to the sub-application's space, so if they try to go outside that bounding area, they are clipped.

Deep linking in sandboxed applications

The BrowserManager controls deep linking support in applications built with Flex. It is a singleton within its security domain. Sub-applications in sibling application domains cannot access the main application's BrowserManager, regardless of whether a sub-application is trusted or untrusted. As a result, a sub-application in a separate security domain or application domain cannot modify the URL nor can it access the URL.

If the sub-application is untrusted, do not give it access to the URL. If the sub-application is trusted (such as with a multi-versioned application that is not sandboxed), you can write custom code that handles the interaction between the sub-application and the main application's BrowserManager. Typically, you call this method in the main application that accesses the URL, or create an interface that acts as a gatekeeper for this interaction.

If you try to get an instance of the BrowserManager from within a sub-application, Flash Player throws an error.

Dragging and dropping in sandboxed applications

When the sub-application is in a different security domain from the main application, the user cannot drag data between the applications. The DragProxy lets the user drag an item to the edge of the sub-application's area of the screen. At that point, the mouse cursor changes back to whatever mouse cursor is correct for the new security domain.

The proxy for the item stays at the boundary of the sub-application, and might be clipped.

Listening for mouse events with sandboxed applications

Mouse interaction between sub-applications and main applications can be confusing, especially when those events occur in different application domains. This interaction is further muddied when the applications are in different security domains. To listen for mouse events outside the security domain, you use the SandboxMouseEvent object. You can listen for this event in a main application for a mouse event that is triggered from the sub-application, and vice versa.

The following application is like the application in the topic, "Listening for mouse events in multi-versioned applications" on page 216, with some exceptions. For example, while the `MouseEvent.MOUSE_MOVE` and `MouseEvent.MOUSE_UP` events are registered, the `SandboxMouseEvent.MOUSE_UP_SOMEWHERE` event is also registered. This event can be registered by any SystemManager within the security domain. All applications can then receive notification if this event is triggered. To get the mouse position, this application uses the `globalToLocal()` method. You can see how to determine the absolute position of an object in a sub-application when you don't have access to the stage.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/ZoomerPattern4.mxml -->
<s:Application
    creationComplete="setup()"
    height="250"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <fx:Script>
     <![CDATA[
          import mx.core.UIComponent;
          import mx.events.SandboxMouseEvent;
          import mx.managers.PopUpManager;
          [Bindable]
          public var data1:Array = ["Ice Cream", "Fudge", "Whipped Cream", "Nuts"];
          public var zoomTool:UIComponent;
          public function setup():void {
               // Draw the zoom rectangle.
               zoomWidget.graphics.lineStyle(1);
               zoomWidget.graphics.beginFill(0, 0);
               zoomWidget.graphics.drawRect(0, 0, 17, 17);
               zoomWidget.graphics.endFill();

               // Listen for mouse down events.
               zoomWidget.addEventListener(MouseEvent.MOUSE_DOWN, zoom_mouseDownHandler);
          }
          private var lastX:int;
          private var lastY:int;
          private function zoom_mouseDownHandler(event:MouseEvent):void {
               // When the mouse is down, listen for the move and up events.
               // The getSandboxRoot() method lets you listen to all mouse activity in your
               // SecurityDomain.

               systemManager.getSandboxRoot().addEventListener(
                    MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
               systemManager.getSandboxRoot().addEventListener(
                    MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
               // The SandboxMouseEvents provide you with some mouse information,
               // but not its position
               systemManager.getSandboxRoot().addEventListener(
                    SandboxMouseEvent.MOUSE_UP_SOMEWHERE, zoom_mouseUpHandler);

               // Update last position of the mouse.
               lastX = event.stageX;
               lastY = event.stageY;
              // Create and pop up the zoomTool. This is the rectangle that is dragged around.
               // It must be a popup so that it can float over other content.
               zoomTool = new UIComponent();
               PopUpManager.addPopUp(zoomTool, this);

               var pt:Point = new Point(zoomWidget.transform.pixelBounds.x,
                    zoomWidget.transform.pixelBounds.y);
               pt = zoomTool.parent.globalToLocal(pt);
               zoomTool.x = pt.x;
               zoomTool.y = pt.y;
               zoomTool.graphics.lineStyle(1);
               zoomTool.graphics.beginFill(0, 0);
               zoomTool.graphics.drawRect(0, 0, 17, 17);
               zoomTool.graphics.endFill();
               // Hide the rectangle that was the target.
               zoomWidget.visible = false;
          }
          private function zoom_mouseMoveHandler(event:MouseEvent):void {
               // Update the position of the dragged rectangle.
               zoomTool.x += event.stageX - lastX;
               zoomTool.y += event.stageY - lastY;
               lastX = event.stageX;
               lastY = event.stageY;
               var bm:BitmapData = new BitmapData(16, 16);

               // Capture the bits on the screen.
               // Use the globalToLocal() method to get the coordinates of the rectangle.
               // Untrusted sub-applications do not have access to the stage so you have
               // to call the globalToLocal() method on a point.
               var pt:Point = new Point(zoomTool.transform.pixelBounds.x + 2,
                    zoomTool.transform.pixelBounds.y + 2);
               pt = DisplayObject(systemManager.getSandboxRoot()).globalToLocal(pt);
               bm.draw(DisplayObject(systemManager.getSandboxRoot()),
                    new Matrix(1, 0, 0, 1, -pt.x, -pt.y));
               // Create a Bitmap to hold the bits.
               if (zoomed.numChildren == 0) {
                    var bmp:Bitmap = new Bitmap();
                    zoomed.addChild(bmp);
               } else
                    bmp = zoomed.getChildAt(0) as Bitmap;
               // Set the bits.
               bmp.bitmapData = bm;
               // Zoom in on the bits.
               bmp.scaleX = bmp.scaleY = 8;
          }
          private function zoom_mouseUpHandler(event:Event):void {
               // Remove the listeners.
               systemManager.getSandboxRoot().removeEventListener(
                    MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
               systemManager.getSandboxRoot().removeEventListener(
                    MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
               systemManager.getSandboxRoot().removeEventListener(
                    SandboxMouseEvent.MOUSE_UP_SOMEWHERE, zoom_mouseUpHandler, true);
               // Replace the target rectangle.
               zoomWidget.visible = true;

               // Remove the dragged rectangle.
```

```
                PopUpManager.removePopUp(zoomTool);
            }
     ]]>
     </fx:Script>

    <mx:HBox>
          <mx:HBox backgroundColor="0x00eeee" height="140" paddingTop="4" paddingRight="4">
               <mx:Label text="Drag Rectangle"/>
               <mx:UIComponent id="zoomWidget" width="17" height="17"/>
               <mx:Canvas id="zoom"
                    borderStyle="solid"
                    width="132"
                    height="132"
               >
                    <mx:UIComponent id="zoomed" width="128" height="128"/>
               </mx:Canvas>
          </mx:HBox>
          <mx:List dataProvider="{data1}"/>
     </mx:HBox>
</s:Application>
```

The following example main application loads the previous example application. It sets the value of the `trustContent` property to `false` to mimic the behavior of a remotely loaded untrusted sub-application. It also links the PopUpManager because the sub-application uses it.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainZoomerPattern4.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <fx:Script>
          <![CDATA[
               /* The PopUpManager must be linked in to all applications that are
                  at the same security sandbox root. Because the sub-application
                  in this example uses the PopUpManager, the main application must also
                  link it it. */
               import mx.managers.PopUpManager; PopUpManager;
          ]]>
     </fx:Script>
     <mx:Text text="Portal (untrusted versioning application):"/>

     <mx:SWFLoader id="swf1"
          loadForCompatibility="true"
          trustContent="false"
          source="ZoomerPattern4.swf"/>

</s:Application>
```

Accessing flashVars variables in sandboxed applications

You can access application parameters that were passed into the sub-application as `flashVars` variables. To do this, you access the Array of parameters on the application object by using the `getSandboxRoot()` method.

The following example gets a reference to the root application and accesses the parameters:

```
var app:DisplayObject = DisplayObject(
    SystemManager.getSandboxRoot()["application"]);
var parameters:Object = app["parameters"];
```

Developing multi-versioned applications

In most applications, even very large applications, the entire base of source code is compiled before release. This recompilation ensures that all pieces of the applications are using the same APIs. This model works well for desktop software and software whose releases are tightly controlled. But it does not always work well in the Internet model of application development, where applications are continually updated.

Internet applications are often developed and released gradually, with new pieces of functionality being added to the application all the time. To ease application development of this kind, Flex lets you create multi-versioned applications. A single application can load multiple sub-applications even if those sub-applications are compiled with different versions of the Flex framework. Sub-applications must, however, be compiled with the same or older version of the compiler that was used to compile the main application.

Using multi-versioned applications is common for sandboxed applications where more than one group of developers works on different pieces of functionality. It is also common on very large applications that are continually improved on and released.

When compiling multi-versioned applications, you should include the MarshallingSupport class into the main application and sub-applications. You do this with the `includes` compiler argument, as the following example shows:

```
-includes=mx.managers.systemClasses.MarshallingSupport
```

Using RPC and DataServices-related functionality in your multi-versioned applications has some limitations. These limitations depend on the type of remote data access you use: RPC classes with a proxy server, RPC classes without a proxy server, and DataServices and RemoteObject functionality. In many of these cases, you should develop sandboxed applications rather than multi-versioned applications. For more information, see “Using RPC and DataServices classes with multi-versioned applications” on page 214.

Flex supports multi-versioned applications that were compiled with version 3.2 of the Flex framework and later. For example, your main application can be a Flex 4.6 application, and the loaded applications can be compiled with the following versions of the framework:

- 3.2
- 3.4
- 3.6
- 4
- 4.5
- 4.6

If you try to load an sub-application that was compiled with a more recent version of the framework than the main application, you will get run time errors if the two applications use a shared resource whose API changed between versions of the Flex framework.

In general, a main application that is compiled for Flash Player 11 can load sub-applications that were compiled for Flash Player 11 or Flash Player 10. However, main applications that were compiled for Flash Player 10 cannot load sub-applications that were compiled for Flash Player 11.

Sub-applications that are compiled with an older version of the Flex framework can be loaded either locally (from the same domain or subdomain, in which case they are trusted) or cross-domain (from a different domain, in which case they are untrusted by default). A trusted multi-versioned application will have nearly the same level of interoperability with the main application as a single-versioned trusted application. Styles and resource bundles are not shared, but otherwise they have the same level of interoperability. An untrusted multi-versioned application has slightly more interoperability than a single-versioned untrusted application has when loaded. With untrusted multi-versioned applications, focus management and mouse events work with the main application.

When you load a multi-versioned application, you set the `loadForCompatibility` property of the SWFLoader control to `true`. Setting this property instructs the loader to load the application into a sibling application domain. Applications that are in sibling application domains each maintain their own class definitions. They communicate through event passing, if necessary. For example, if a sub-application must do something that only the top-level application's manager can do, then it communicates with the main application by dispatching an event.

The following sections describe how to develop multi-versioned applications. They assume that all applications are trusted (loaded into the same security domain), and loaded into sibling application domains of the main application. For information about building applications that are in separate security domains, see "Developing sandboxed applications" on page 205.

Using RPC and DataServices classes with multi-versioned applications

RPC and DataServices classes are a special case when used in multi-versioned sub-applications. All applications in the same security domain must share the same definitions of the RPC and DataServices classes for messaging to work. In addition, they might also need to share the same definitions for custom value object classes. This can be a problem because the only way to ensure that a main application and a sub-application use the same set of definitions is to load the sub-application into a child application domain of the main application, which would prevent multi-versioned applications from working. As a result, to use RPC classes in multi-versioned applications, you can do one of the following:

Load the applications as sandboxed applications If you do this, you must call the `Security.allowDomain()` method from the main application on the sub-application's domain in the SWFLoader control's `complete` event handler. You must also call the `Security.allowDomain()` method from the sub-application on the main application's domain. This provides the same level of application interoperability as multi-versioned applications. In the sub-application, you must call the `allowDomain()` method early on in the initialization cycle. Use the application's `preinitialize` event.

Use a bootstrap loader You define the messaging classes in a bootstrap loader. You then load the main application into the bootstrap loader. The main application and its sub-applications can then share those class definitions.

When you use RPC classes (such as WebService and HTTPService) without a proxy server, you do not need to load the sub-application as a sandboxed application or externalize the classes in a bootstrap class loader. Applications that use these classes without a proxy server should work as well as any multi-versioned application. Be sure to that either the URLs for the target services are on the same server as the main application, or that the target server has a crossdomain.xml file on it that allows access.

More Help topics

"Developing sandboxed applications" on page 205

"Bootstrap loading" on page 219

Pop-up controls in multi-versioned applications

When a sub-application launches a pop-up control, it floats over the entire application, and can be dragged anywhere on the stage. Pop-up controls are created in the sub-application’s application domain and passed to the main application's PopUpManager. Pop-up controls cannot be strongly-typed as IUIComponent, so they are marshaled from the sub-application to the main application for display.

Pop-up windows behave as follows in a multi-versioned sub-application (these behaviors are the same for single-versioned sub-applications):

- Launching a modal dialog box dims the entire application, not just the sub-application that launched it
- Centering a pop-up centers it over the entire application, not just the sub-application
- Dragging pop-up controls works over the entire application, not just the sub-application
- Focus shifts to the pop-up when you first launch a pop-up from the sub-application

The PopUpManager must be linked in to all applications that are sandbox roots. When using the PopUpManager in a child application, there must also be an instance of the PopUpManager in the main application. To link the PopUpManager to a main application that doesn’t use it, you can add the following code:

```
import mx.managers.PopUpManager;
private var popupManager1:PopUpManager;
```

If you do not link a PopUpManager into an application, and a sub-application in its sandbox requests that it display a modal dialog box, you might get the following error:

```
No class registered for interface 'mx.managers::IPopUpManager'.
```

List controls require some additional code when used in sub-applications. If you have a List control in a sub-application’s pop-up and no List control in your main application, you must also link the List class into your main application. In addition, the List control must have a data provider. You can link an invisible List into your main application and set its data provider to an empty array, as the following example shows:

```
<mx:List visible="false" includeInLayout="false" dataProvider="[]"/>
```

Embedded fonts in multi-versioned applications

Applications that are in the same application domain are able to use the same embedded fonts by specifying the font name. However, if the sub-application is loaded into a different application domain (as is the case with multi-versioned applications), then the sub-application must embed the font to use it.

Styles and style modules in multi-versioned applications

Using styles and style modules in multi-versioned sub-applications is the same as using them in sandboxed sub-applications. For more information on using styles and style modules in sub-applications, see “Styles and style modules in sandboxed applications” on page 206.

Focus in multi-versioned applications

Using the FocusManager in multi-versioned sub-applications is the same as using the FocusManager in sandboxed sub-applications. (See “Focus in sandboxed applications” on page 207.)

Cursors in multi-versioned applications

The CursorManager class in the main application and sub-application communicate much like the PopupManager class. If the sub-application changes the cursor to a custom cursor, this custom cursor continues to appear when the cursor is moved outside the sub-application and over the main application. Calling a method or setting a property on the CursorManager in the sub-application bubbles up to the main application, and vice versa.

If a busy cursor is visible while the mouse is over the sub-application, and you move the cursor to the main application, the cursor remains as a busy cursor.

Localizing multi-versioned applications

Using resource bundles in multi-versioned sub-applications is the same as using them in sandboxed sub-applications. For more information on using resource bundles in sub-applications, see "Localizing sandboxed applications" on page 208.

ToolTip objects in multi-versioned applications

ToolTip objects are created in the sub-application's application domain and are passed to the main application's ToolTipManager. This is an example of marshaling a class across application boundaries.

This behavior applies to error tips and data tips on List objects in multi-versioned applications as well.

Layouts in multi-versioned applications

The main application dictates the area of the screen that is available to the sub-application for screen layout.

LayoutManagers in different applications run independent of one another. The applications are not clipped.

Deep linking in multi-versioned applications

You cannot access the main application's BrowserManager directly from a sub-application that is in a different application domain. The sub-application's BrowserManager is disabled. This is the same for sandboxed and multi-versioned sub-applications. For more information, see "Deep linking in sandboxed applications" on page 209.

Dragging and dropping in multi-versioned applications

When the sub-application is in the same security domain as the main application, but in a different application domain, you can drag list items from a sub-application to a list in the main application, and vice versa. For this to work across applications, the class that defines the drop target must be public.

The DragProxy maintains the item's appearance during the entire event, regardless of which application the mouse is over. Drag events are triggered for all source and destination controls, as if they are in the same application.

In the background, the DragManager generates the DragProxy in the sub-application's application domain then passes that DragProxy to the main application's DragManager during the drag operation.

Strongly-typed objects cannot be passed from the main application to the sub-application or vice versa. As a result, the DragManager accepts a DragProxy that is not strongly typed.

All properties and methods defined by the IDragManager interface are handled through delegation to the main application's DragManager.

If you drag a generic type object, the object will be of generic type Object in Flash Player's application domain. Flex marshals the properties so that the drag and drop operation works without having to write any custom code.

If you use a custom class as part of the drop data, that class cannot be shared as strongly-typed. In that case, override the drop events and add logic to marshall the data from the drag source to the drop target.

Listening for mouse events in multi-versioned applications

Mouse interaction between a main application and a sub-application can be confusing, especially when those events occur in different application domains. For example, when you click and drag an object in a sub-application, and then release the button over the main application, typical Flex mouse events such as `MOUSE_UP`, `MOUSE_DOWN_OUTSIDE`, or `MOUSE_LEAVE` are triggered but cannot be listened to directly in the sub-application.

To listen for mouse events across application domains, listen to all mouse activity in the security domain. To listen, get a reference to the sandbox root and register your event listeners with that SystemManager.

When loaded into a separate application domain, the `topLevelSystemManager` property refers to the sub-application's SystemManager because the main application's SystemManager is in another application domain. As a result, you do not use the `topLevelSystemManager` property to get a reference to the main application's SystemManager. Instead, you use the `systemManager.getSandboxRoot()` method to get a reference to the top-level SystemManager in the security domain. From this SystemManager, you can access all mouse activity in the current security domain.

The following example uses calls to the `getSandboxRoot()` method to get references to the top-level SystemManager in the security domain.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/ZoomerPattern3.mxml -->
<s:Application
    creationComplete="setup()"
    height="250"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <fx:Script>
     <![CDATA[
          import mx.core.UIComponent;
          import mx.managers.PopUpManager;
          [Bindable]
          public var data1:Array = ["Ice Cream", "Fudge", "Whipped Cream", "Nuts"];
          public var zoomTool:UIComponent;
          public function setup():void {
               // Draw the zoom rectangle.
               zoomWidget.graphics.lineStyle(1);
               zoomWidget.graphics.beginFill(0, 0);
               zoomWidget.graphics.drawRect(0, 0, 17, 17);
               zoomWidget.graphics.endFill();

               // Listen for mouse down events.
               zoomWidget.addEventListener(MouseEvent.MOUSE_DOWN, zoom_mouseDownHandler);
          }
          private var lastX:int;
          private var lastY:int;
          private function zoom_mouseDownHandler(event:MouseEvent):void {
               // When the mouse is down, listen for the move and up events.
               // The getSandboxRoot() method lets you listen to all mouse activity in your
               // SecurityDomain.
               systemManager.getSandboxRoot().addEventListener(
                    MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
               systemManager.getSandboxRoot().addEventListener(
                    MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
               // Update last position of the mouse.
               lastX = event.stageX;
               lastY = event.stageY;
              // Create and pop up the zoomTool. This is the rectangle that is dragged around.
               // It must be a popup so that it can float over other content.
               zoomTool = new UIComponent();
               PopUpManager.addPopUp(zoomTool, this);

               var pt:Point = new Point(zoomWidget.transform.pixelBounds.x,
                    zoomWidget.transform.pixelBounds.y);
               pt = zoomTool.parent.globalToLocal(pt);
               zoomTool.x = pt.x;
               zoomTool.y = pt.y;
               zoomTool.graphics.lineStyle(1);
               zoomTool.graphics.beginFill(0, 0);
               zoomTool.graphics.drawRect(0, 0, 17, 17);
               zoomTool.graphics.endFill();
               // Hide the rectangle that was the target.
               zoomWidget.visible = false;
          }
          private function zoom_mouseMoveHandler(event:MouseEvent):void {
               // Update the position of the dragged rectangle.
               zoomTool.x += event.stageX - lastX;
               zoomTool.y += event.stageY - lastY;
               lastX = event.stageX;
               lastY = event.stageY;
               var bm:BitmapData = new BitmapData(16, 16);
               // Capture the bits on the screen.
               // You must use the getSandboxRoot() method here, too, because it gets
               // the parent of all of the pixels you are allowed to access.
               bm.draw(DisplayObject(systemManager.getSandboxRoot()),
                    new Matrix(1, 0, 0, 1, -zoomTool.transform.pixelBounds.x - 2,
                    -zoomTool.transform.pixelBounds.y - 2));
               // Create a Bitmap to hold the bits.
               if (zoomed.numChildren == 0) {
                    var bmp:Bitmap = new Bitmap();
                    zoomed.addChild(bmp);
               } else
                    bmp = zoomed.getChildAt(0) as Bitmap;
               // Set the bits.
               bmp.bitmapData = bm;
               // Zoom in on the bits.
               bmp.scaleX = bmp.scaleY = 8;
          }
          private function zoom_mouseUpHandler(event:Event):void {
               // Remove the listeners.
               systemManager.getSandboxRoot().removeEventListener(
                    MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
               systemManager.getSandboxRoot().removeEventListener(
                    MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
               // Replace the target rectangle.
               zoomWidget.visible = true;
               // Remove the dragged rectangle.
```

```
                PopUpManager.removePopUp(zoomTool);
            }
        ]]>
    </fx:Script>

    <mx:HBox>
        <mx:HBox backgroundColor="0x00eeee" height="140" paddingTop="4" paddingRight="4">
            <mx:Label text="Drag Rectangle"/>
            <mx:UIComponent id="zoomWidget" width="17" height="17"/>
            <mx:Canvas id="zoom"
                borderStyle="solid"
                width="132"
                height="132"
            >
                <mx:UIComponent id="zoomed" width="128" height="128"/>
            </mx:Canvas>
        </mx:HBox>
        <mx:List dataProvider="{data1}"/>
    </mx:HBox>
</s:Application>
```

The following example main application loads the previous example application. It sets the `loadForCompatibility` property to `true` so that the sub-application loads a multi-versioned application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainZoomerPattern3.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:Text text="Cross-Versioning (trusted versioning application):"/>
    <mx:SWFLoader id="swf1" loadForCompatibility="true" source="ZoomerPattern3.swf"/>

</s:Application>
```

Bootstrap loading

A bootstrap loader is a lightweight application that defines classes that you want to share among applications.

The bootstrap loader gets loaded first instead of the actual main application, and then loads the main application into a child application domain of itself. The main application then loads a sub-application into a sibling application domain. The two applications are both loaded into child application domains of the bootstrap loader's application domain. The result is that they share any classes that are defined in the bootstrap, but use their own class definitions for everything else.

You can typically only have one bootstrap loader in a security domain. Bootstrapped classes are not shared across security domains.

Furthermore, when using bootstrap loading to load sub-applications, the sub-applications must be compiled with the same or older version of the compiler that the main application is compiled with.

Changes to any class in a bootstrap loader can break other sub-applications. As a result, ensure that the classes you add to a bootstrap loader are not likely to change.

When you use RPC classes with a proxy server (such as Blaze DS or LCDS), you must externalize the RPC classes in a bootstrap loader for multi-versioned applications. The bootstrap loader should contain only the RPC classes. You typically do not add your custom value object classes to the bootstrap loader to get or put data on the server. You can add those classes if you want to reduce marshalling, but in that case they should be simple classes that do not link in Flex framework classes such as Array or ArrayCollection.

When you use DataServices or RemoteObject functionality, you typically use a bootstrap loader for loading multi-versioned applications. The bootstrap loader should contain the necessary RPC and DataServices-related classes. It must also contain any value object classes that are used to send data to or from the server. If you add custom value object classes to your bootstrap loader, they should be simple classes that do not link in Flex framework classes such as Array or ArrayCollection. Adding complex value object classes to the bootstrap loader is not supported.

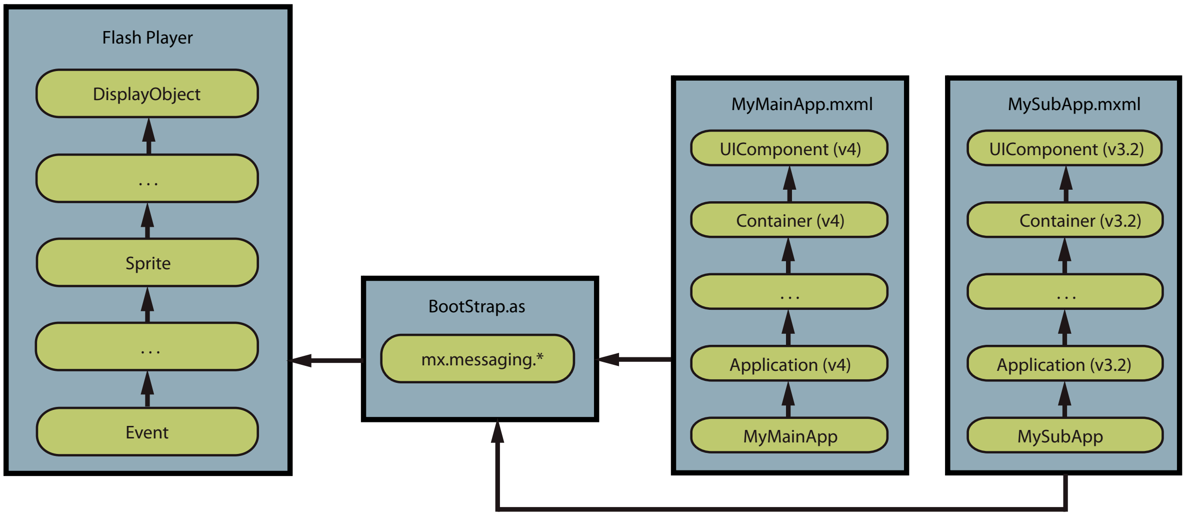
The following is an example of the bootstrap loader for RPC classes. This kind of bootstrap loader is commonly used when a sub-application uses classes such as HTTPService and WebService with a proxy server.

Take note of the way that the MainApp.swf is loaded. The bootstrap loader uses a Loader class, and does not define the LoaderContext, which results in default behavior for the Loader. The defaults are to load a sub-application into a child application domain, and to assume the defaults for the security domain.

Notice, too, that for each import, a definition of the class immediately follows the class's import statement exists. The result is that the class definitions are linked into the bootstrap loader. The loaded application, and all subsequently loaded applications, then share those definitions.

```
// apploading/RPCBootstrapLoader.as
package {
import flash.display.Loader;
import flash.display.Sprite;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.Event;
import flash.net.URLRequest;
import flash.system.ApplicationDomain;
/**
 *  Classes used by the networking protocols go here. These are the classes
 *  whose definitions are added to the bootstrap loader and then shared
 *  by the main application and all sub-applications.
 */
import mx.messaging.config.ConfigMap; ConfigMap;
import mx.messaging.messages.AcknowledgeMessage; AcknowledgeMessage;
import mx.messaging.messages.AcknowledgeMessageExt; AcknowledgeMessageExt;
import mx.messaging.messages.AsyncMessage; AsyncMessage;
import mx.messaging.messages.AsyncMessageExt; AsyncMessageExt;
import mx.messaging.messages.CommandMessage; CommandMessage;
import mx.messaging.messages.CommandMessageExt; CommandMessageExt;
import mx.messaging.messages.ErrorMessage; ErrorMessage;
import mx.messaging.messages.HTTPRequestMessage; HTTPRequestMessage;
import mx.messaging.messages.MessagePerformanceInfo; MessagePerformanceInfo;
import mx.messaging.messages.RemotingMessage; RemotingMessage;
import mx.messaging.messages.SOAPMessage; SOAPMessage;
import mx.messaging.channels.HTTPChannel; HTTPChannel;
import mx.core.mx_internal;
[SWF(width="600", height="700")]
public class RPCBootstrapLoader extends Sprite {
        /**
         *  The URL of the application SWF that this bootstrap loader loads.
         */
        private static const applicationURL:String = "MainApp.swf";
        /**
         *  Constructor.
         */
        public function RPCBootstrapLoader() {
                super();
                if (ApplicationDomain.currentDomain.hasDefinition("mx.core::UIComponent"))
                       throw new Error("UIComponent should not be in the bootstrap loader.");
                stage.scaleMode = StageScaleMode.NO_SCALE;
                stage.align = StageAlign.TOP_LEFT;
                if (!stage)
                        isStageRoot = false;
                root.loaderInfo.addEventListener(Event.INIT, initHandler);
        }
        /**
         *  The Loader that loads the main application's SWF file.
         */
        private var loader:Loader;
        /**
         *  Whether the bootstrap loader is at the stage root or not.
         *  It is only the stage root if it was the root
         *  of the first SWF file that was loaded by Flash Player.
         *  Otherwise, it could be a top-level application but not stage root
         *  if it was loaded by some other non-Flex shell or is sandboxed.
         */
        private var isStageRoot:Boolean = true;
        /**
         *  Called when the bootstrap loader's SWF file has been loaded.
         *  Starts loading the application SWF specified by the applicationURL property.
         */
        private function initHandler(event:Event):void {
                loader = new Loader();
                addChild(loader);
                loader.load(new URLRequest(applicationURL));
                loader.addEventListener("mx.managers.SystemManager.isBootstrapRoot",
                        bootstrapRootHandler);
                loader.addEventListener("mx.managers.SystemManager.isStageRoot",
                        stageRootHandler);
                stage.addEventListener(Event.RESIZE, resizeHandler);
        }
        private function bootstrapRootHandler(event:Event):void {
                // Cancel event to indicate that the message was heard.
                event.preventDefault();
        }
        private function stageRootHandler(event:Event):void {
                // Cancel event to indicate that the message was heard.
                if (!isStageRoot)
                        event.preventDefault();
        }
        private function resizeHandler(event:Event):void {
                loader.width = stage.width;
                loader.height = stage.height;
                Object(loader.content).setActualSize(stage.stageWidth, stage.stageHeight);
        }
}
}
```

The following image shows where each domain gets it class definitions for a bootstrap loader that defines the mx.messaging.* classes.



Flex Ajax Bridge

The Adobe® Flex™ Ajax Bridge (FABridge) is a small, unobtrusive code library that you can insert into an Flex application, a Flex component, or an empty SWF file to expose it to scripting in the web browser.

About the Flex Ajax Bridge

The Flex Ajax Bridge (FABridge) is a small code library that you can insert into an Flex application, a Flex component, or an empty SWF file to expose it to scripting in the browser.

Rather than having to define new, simplified APIs to expose a graph of ActionScript objects to JavaScript, with FABridge you can make your ActionScript classes available to JavaScript without any additional coding. After you insert the library, essentially anything you can do with ActionScript, you can do with JavaScript.

Adobe Flash Player has the native ability, through the External API (the ExternalInterface class), to call JavaScript from ActionScript, and vice versa. But ExternalInterface has some limitations:

- The ExternalInterface class requires you, the developer, to write a library of extra code in both ActionScript and JavaScript, to expose the functionality of your Flex application to JavaScript, and vice versa.
- The ExternalInterface class also limits what you can pass across the gap – primitive types, arrays, and simple objects are legal, but user-defined classes, with associated properties and methods, are off-limits.
- The ExternalInterface class lets you define an interface so your JavaScript can call your ActionScript. FABridge lets you write JavaScript instead of ActionScript.

When to use the Flex Ajax Bridge

The FABridge library is useful in the following situations:

- You want to use a rich Flex component in an Ajax application but do not want to write a lot of Flex code. If you wrap the component in a FABridge-enabled stub application, you can script it entirely from JavaScript, including using JavaScript generated remotely by the server.

- You have only one or two people on your team who know Flex. The FABridge library lets everyone on your team use the work produced by one or two Flex specialists.
- You are building an integrated rich Internet application (RIA) with Flex and Ajax portions. Although you could build the integration yourself using ExternalInterface, you might find it faster to start with the FABridge.

Requirements for using the Ajax Bridge

To use the FABridge library and samples, you must have the following:

- Flex Ajax Bridge, which is included in the following directory of the Flex 3 SDK installation:

  installation_dir\frameworks\javascript\fabridge
- Adobe Flex SDK
- Adobe® Flash® Player 9 or later, or Adobe® AIR™ 1.5 or later
- Microsoft Internet Explorer, Mozilla Firefox, or Opera with JavaScript enabled
- Any HTTP server to run the samples

To run the samples:

1. Place the *installation_dir*\frameworks\javascript\fabridge\src and *installation_dir*\frameworks\javascript\fabridge\samples folders side by side on any HTTP server.
2. Open a web browser to http://*yourwebserver*/samples/FABridgeSample.html and samples/SimpleSample.html and follow the instructions there. Make sure you access the samples through http:// URLs and not file:// URLs. The Flash Player security sandbox prevents them from working correctly when they are accessed as local files.

Integrating with the Flex Ajax Bridge

To use the FABridge library in your own Flex and Ajax applications in Adobe® Flash® Builder™, right-click a project in the Flex Navigator and select Create Ajax Bridge. For more information, see Automatically generating Flex Ajax Bridge code.

Or, complete the following manual steps:

1. Add the *installation_dir*\frameworks\javascript\fabridge\src folder to the ActionScript classpath of your Flex application.
2. If you are compiling from the command line, add the *installation_dir*\frameworks\javascript\fabridge\src folder to your application by specifying the `--actionscript-classpath` compiler option.
3. Add the following tag to your application file:

   ```
   <s:Application …>
       <fab:FABridge xmlns:fab="bridge.*" />
   ...
   ```

Use the following code to access your application instance from JavaScript:

```
function useBridge() {
    var flexApp = FABridge.flash.root();
}
```

To get the value of a property, call it like a function, as the following example shows:

```
function getMaxPrice() {
    var flexApp = FABridge.flash.root();
    var appWidth = flexApp.getWidth();
    var maxPrice = flexApp.getMaxPriceSlider().getValue();
}
```

To set the value of a property from JavaScript, call the `setPropertyName()` function, as the following example shows:

```
function setMaxPrice(newMaxPrice) {
    var flexApp = FABridge.flash.root();
    flexApp.getMaxPriceSlider().setValue(newMaxPrice);
}
```

You call object methods directly, just as you would from ActionScript, as the following example shows:

```
function setMaxPrice(newMaxPrice) {
    var flexApp = FABridge.flash.root();
    flexApp.getShoppingCart().addItem("Antique Figurine", 12.99);
}
```

You also pass functions, such as event handlers, from JavaScript to ActionScript, as the following example shows:

```
function listenToMaxPrice() {
    var flexApp = FABridge.flash.root();
    var maxPriceCallback = function(event) {
        document.maxPrice = event.getNewValue();
        ocument.loadFilteredProducts(document.minPrice, document.maxPrice);
    }
    flexApp.getMaxPriceSlider().addEventListener("change", maxPriceCallback);
}
```

To run initialization code on a Flex file, you must wait for it to download and initialize first. Register a callback to be invoked when the movie is initialized, as the following example shows:

```
function initMaxPrice(maxPrice) {
    var initCallback = function() {
        var flexApp = FABridge.flash.root();
        flexApp.getMaxPriceSlider().setValue(maxPrice);
    }
    FABridge.addInitializationCallback("flash",initCallback);
}
```

To script multiple applications on the same page, give them unique bridge names through the flashvars mechanism. Use the bridge name to access them from the bridge, and to register for initialization callbacks, as the following example shows:

```
<object ...>
    <param name='flashvars' value='bridgeName=shoppingPanel'/>
    <param name='src' value='app.swf'/>
    <embed ... flashvars='bridgeName=shoppingPanel'/>
</object>
function initMaxPrice(maxPrice) {
    var initCallback = function() {
        var flexApp = FABridge.shoppingPanel.root();
        flexApp.getMaxPriceSlider().setValue(maxPrice);
    }
    FABridge.addInitializationCallback("shoppingPanel",initCallback);
}
```

Automatic memory management

The FABridge provides automatic memory management that uses a reference counting mechanism for all objects that are passed across the bridge. Objects created from the JavaScript side are kept in memory unless the memory is manually released. Events and other ActionScript-initiated objects are destroyed as soon as the corresponding JavaScript function that handles them directly completes its execution. You manually call the `addRef()` method for an object to have it remain available, or call the `release()` method to decrease its reference counter.

If you must break the function call chain by using the `setTimeout()` function in JavaScript, (for example to act on an event later on, as the following example shows), you must ensure that the event will still exist. Because the FABridge implements a reference counting mechanism to save memory, events thrown from ActionScript exist only for the duration of the dispatch function.

```
var flexApp = FABridge.flash.root();
flexApp.getMaxPriceSlider().addEventListener("change", maxPriceCallback );
function maxPriceCallback(event) {
    // When the doSomethingLater function is hit, the event is no longer available;
    // to make it work you would have to call FABridge.addRef(event); then, when you're
    // done with it, call FABridge.release(event).
    setTimeout(function() {doSomethingLater(event);},10);
}
```

Manually destroying objects

You can manually destroy a specific object that has been passed across the bridge, regardless of its reference count by invoking the `releaseNamedASObject(myObject)` method from JavaScript. This invalidates the object over the bridge and any future calls to it or one of its methods will throw an error.

Handling exceptions

Exceptions that take place in the ActionScript of the bridge as a direct consequence of some JavaScript action are now thrown over the bridge into JavaScript. The mechanism works as follows:

- When an exception is raised in the ActionScript section, it is caught in a try-catch block, serialized, and passed to JavaScript.
- When the JavaScript part receives an answer from ActionScript, it checks for the exception serialization and, if found, throws a JavaScript error with the message received from ActionScript.

***Note:** To catch and use the exception information, you must surround the code that calls into ActionScript with a `try-catch` block. You can handle the error in the `catch(e)` block.*

Limitations of the Flex Ajax Bridge

The FABridge library has been tested on Mozilla Firefox 2 (Windows and Linux), Microsoft Internet Explorer 6, Opera 9, and Apple Safari 2.0.4.

Exceptions thrown across the bridge into JavaScript depend on the user having installed Flash Debug Player to display the entire error description. Otherwise, only the error ID is thrown.

For performance reasons, when an anonymous object is sent from ActionScript to JavaScript, the bridge assumes it contains only primitives, arrays, and other anonymous objects, and no strongly typed objects or methods. Instances or methods sent as part of an anonymous object are not bridged correctly.

Communicating with the wrapper

You exchange data between an Adobe® Flex® application and the HTML page that embeds that application in several ways, depending on the type of integration that your application requires.

About exchanging data with applications

Applications built with Flex generally exist inside larger web applications that control everything from security to state management to the overall look and feel of the website. In this scenario it is important that the application is able to communicate with the surrounding environment, providing a deeper integration with the larger web application. Enabling the application to communicate with the environment provides a method of integration with other technologies such as Ajax.

Often, an application is loaded in a browser within a wrapper. This wrapper is often an HTML page that can include JavaScript or other client-side logic that the application can interact with.

There are several ways to communicate between the surrounding environment and the application; depending on the type of integration required, any combination of `flashVars` properties, query string parameters, the `navigateToURL()` method, and the ExternalInterface class can be employed. In addition, you can communicate between two applications on the same page by using SharedObjects.

To pass request data into your applications, you can define `flashVars` properties inside your HTML wrapper and access their values using the `FlexGlobals.topLevelApplication.parameters`. For more information, see "Passing request data with flashVars properties" on page 229. Using these techniques, you can personalize an application without requiring a recompilation.

Use the methods of the ExternalInterface API to call the methods of your applications and vice versa. The addCallback() method exposes methods of your application to the wrapper. The call() method invokes a method within the wrapper and returns any results. If the wrapper is HTML, the `addCallback()` and `call()` methods enable method invocation between your application and the hosted scripting language running within the browser. For more information, see "About the ExternalInterface API" on page 227.

In some situations, you want to open a new browser window or navigate to a new location. You can do this with the navigateToURL() global function. Although it is not part of the ExternalInterface API, this method is flexible enough to let you write JavaScript inside it, and invoke JavaScript functions on the resulting HTML page. The `navigateToURL()` method is a global function in the flash.net package.

More Help topics

"Creating a wrapper" on page 2552

Determining the application's URL

Flex provides some simple mechanisms for getting information about the browser and the environment in which the application runs. From within the application, you can get the URL to the SWF file by using the `FlexGlobals.topLevelApplication.url` property. To access properties of the FlexGlobals class, you must import mx.core.FlexGlobals.

The following example gets the SWF file's URL and extracts the host name from the URL:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/GetURLInfo.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="getHostName()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.core.FlexGlobals;
      [Bindable]
      public var g_HostString:String;
      [Bindable]
      public var g_ContextRoot:String;
      [Bindable]
      public var g_BaseURL:String;
      private function getHostName():void {
         g_BaseURL = FlexGlobals.topLevelApplication.url;
         var pattern1:RegExp = new RegExp("http://[^/]*/");
         if (pattern1.test(g_BaseURL) == true) {
            g_HostString = pattern1.exec(g_BaseURL).toString();
         } else{
            g_HostString = "http://localhost/"
         }
      }
      ]]>
    </fx:Script>

  <mx:Form>
     <mx:FormItem label="Base URL:">
        <mx:Label text="{g_BaseURL}"/>
     </mx:FormItem>
     <mx:FormItem label="Host Name:">
        <mx:Label text="{g_HostString}"/>
     </mx:FormItem>
  </mx:Form>
</s:Application>
```

You can also use the flash.system.Capabilities class to access information about the client, such as Operating System, Player version, and language. For more information, see Using the Capabilities class.

You can access more information about the browser and the application’s environment using the ExternalInterface. For more information, see “About the ExternalInterface API” on page 227.

About the ExternalInterface API

You use the ExternalInterface API to let your application call methods in the wrapper and to allow the wrapper to call functions in your application. The ExternalInterface API consists primarily of the call() and addCallback() methods in the flash.external package.

The following browsers support the ExternalInterface API:

- All versions of Internet Explorer for Windows (5.0 and later)
- Embedded custom ActiveX containers, such as a desktop application embedding the Adobe® Flash® Player ActiveX control

- Any browser that supports the NPRuntime interface (which currently includes the following browsers):
  - Firefox 1.0 and later
  - Mozilla 1.7.5 and later
  - Netscape 8.0 and later
  - Safari 1.3 and later

Before you execute code that uses the ExternalInterface API, you should check whether the browser supports it. You do this by using the `available` property of the ExternalInterface object in your application. The `available` property is a Boolean value that is `true` if the browser supports the ExternalInterface API and `false` if the browser does not. It is a read-only property.

The following example uses the `available` property to detect support for the ExternalInterface API before executing methods that use the class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/CheckExternalInterface.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="checkEI()">
     <s:layout>
        <s:BasicLayout/>
     </s:layout>

     <fx:Script>
        <![CDATA[
         [Bindable]
         public var eiStatus:String;
         private function checkEI():void {
               eiStatus = ExternalInterface.available.toString();
         }
         ]]>
     </fx:Script>

     <s:Label id="l1" text="External Interface supported? {eiStatus}"/>
</s:Application>
```

The `available` property determines only if the browser can support the ExternalInterface API, based on its version and manufacturer. If JavaScript is disabled in the browser, the `available` property still returns `true`.

For examples of using the ExternalInterface API with applications, see “Using the ExternalInterface API to access JavaScript” on page 235 and “Accessing Flex from JavaScript” on page 247.

In addition to requiring that browsers meet certain version requirements, the ExternalInterface API requires that JavaScript is enabled in the browser. You can use the `<noscript>` tag in the HTML page to handle a browser with disabled JavaScript. For more information, see “Handling browsers that disable JavaScript” on page 250.

The ExternalInterface API is restricted by the security sandbox in which the SWF file is running. Its use relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` parameters define. You set the values of the `allowScriptAccess` and `allowNetworking` parameters in the SWF file’s wrapper.

More Help topics

"Creating a wrapper" on page 2552

"Security" on page 117

"About ExternalInterface API security in Flex" on page 252

Enabling Netscape connections

In some versions of Netscape browsers, you must set the value of the `swliveconnect` property to `true` in your HTML wrapper to enable the application to connect with the page's scripting language (usually JavaScript).

If you are using SWFObject 2 to embed your application, then set the value of the `swliveconnect` property on the params object to `true`, as the following example shows:

```
var params = {};
params.swfliveconnect = "true";
params.quality = "high";
params.bgcolor = "#ffffff";
params.allowscriptaccess = "sameDomain";
```

If your wrapper uses the `<object>` and `<embed>` syntax, include `swliveconnect=true` in the `<embed>` tag in your wrapper, as the following example shows:

```
<embed pluginspage='http://www.adobe.com/go/getflashplayer'
    width='300'
    height='100'
    flashvars=''
    src='TitleTest.mxml.swf'
    name='MyApp'
    swliveconnect='true'
/>
```

You are not required to set the value of `swliveconnect` to `true` in the `<object>` tag because the `<object>` tag is used by Microsoft Internet Explorer, and not by Netscape browsers.

Passing request data with flashVars properties

You pass request data to applications built with Flex by defining the `flashVars` properties in the wrapper. You can also access URL fragments by using the BrowserManager. For more information, see "Passing request data with URL fragments" on page 2031.

The `flashVars` properties and URL fragments are read in the application at run time. As a result, changing `flashVars` properties or URL fragments does not require you to recompile the application.

Passing request data with flashVars properties

If you are using the generated wrapper from Flash Builder, then you add `flashVars` variables by creating an object called flashvars, setting properties on that object, and then passing that object to the `swfobject.embedSWF()` method of SWFObject 2.

The following example sets the `firstname` and `lastname` properties of the flashvars object, and then passes that object to the `embedSWF()` method:

```
<script type="text/javascript">
    var swfVersionStr = "10.0.0";
    var xiSwfUrlStr = "playerProductInstall.swf";
    var flashvars = {};
    flashvars.firstname = "Nick";
    flashvars.lastname = "Danger";
    var params = {};
    params.quality = "high";
    params.bgcolor = "#ffffff";
    params.allowscriptaccess = "sameDomain";
    var attributes = {};
    attributes.id = "TestProject";
    attributes.name = "TestProject";
    attributes.align = "middle";
    swfobject.embedSWF(
        "FlashVarTest.swf", "flashContent", "100%", "100%", swfVersionStr,
        xiSwfUrlStr, flashvars, params, attributes);
    swfobject.createCSS("#flashContent", "display:block;text-align:left;");
</script>
```

If your wrapper uses the `<object>` and `<embed>` tags, you can pass variables to your applications by using the `flashVars` properties in the `<object>` and `<embed>` tags in your wrapper. You do this by adding ampersand-separated sets of name-value pairs to these properties.

The following example sets the values of the `firstname` and `lastname` in the `flashVars` properties inside the `<object>` and `<embed>` tags in a simple wrapper:

```
<html>
<head>
<title>code/wrapper/SimplestFlashVarTestWrapper.html</title>
<style>
    body {
        margin: 0px;
        overflow:hidden
    }
</style>
</head>
<body scroll='no'>
<table width='100%' height='100%' cellspacing='0' cellpadding='0'><tr><td valign='top'>
<h1>Simplest FlashVarTest Wrapper</h1>
    <object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' height='100%'
width='100%'>
        <param name='src' value='FlashVarTest.swf'/>
        <param name='flashVars' value='firstname=Nick&lastname=Danger'/>
        <embed name='mySwf' src='FlashVarTest.swf' height='100%' width='100%'
flashVars='firstname=Nick&lastname=Danger'/>
    </object>
</td></tr></table>
</body>
</html>
```

The value of the `flashVars` properties do not have to be static. If you use JSP to return the wrapper, for example, you can use any JSP expression for the value of the `flashVars` properties that can be evaluated to a String.

The following example uses the values stored in the HttpServletRequest object (in this case, you can use form or query string parameters):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<%
    String fName = (String) request.getParameter("firstname");
    String lName = (String) request.getParameter("lastname");
%>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>DynamicFlashVarTestWrapper.jsp</title>
        <script type="text/javascript" src="swfobject.js"></script>

        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            flashvars.firstname = "<%= fName %>";
            flashvars.lastname = "<%= lName %>";
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "FlashVarTest";
            attributes.name = "FlashVarTest";
            attributes.align = "middle";
            swfobject.embedSWF(
                "FlashVarTest.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <body>
        <div id="flashContent"/>
   </body>
</html>
```

You can also PHP expressions to pass query string parameters in a wrapper, as the following example shows:

```
<!-- saved from url=(0014)about:internet -->
<html lang="en">
<head>
<?php
    @ $fName = $_GET['firstname'];
    @ $lName = $_GET['lastname'];
?>
<script type="text/javascript" src="swfobject.js"></script>
<script type="text/javascript">
    var swfVersionStr = "";
    var xiSwfUrlStr = "";
    var flashvars = {};
    flashvars.fName = "<?php echo $fName; ?>"
    flashvars.lName =  "<?php echo $lName; ?>"
    var params = {};
    params.quality = "high";
    params.bgcolor = "#ffffff";
    params.allowscriptaccess = "sameDomain";
    var attributes = {};
    attributes.id = "FlashVarTest";
    attributes.name = "FlashVarTest";
    attributes.align = "middle";
    swfobject.embedSWF(
        "FlashVarTest.swf", "flashContent",
        "100%", "100%",
        swfVersionStr, xiSwfUrlStr,
        flashvars, params, attributes);
    swfobject.createCSS("#flashContent", "display:block;text-align:left;");
</script>
</head>
<body >
    <div id="flashContent">
</body>
</html>
```

For more information about the HTML wrapper, see “Creating a wrapper” on page 2552.

If your user requests the SWF file directly in the browser, without a wrapper, you can access variables on the query string without providing additional code in the wrapper (because there is no wrapper). The following URL passes the name Nick and the hometown San Francisco to the application:

```
http://localhost:8100/flex/myApp.swf?myName=Nick&myHometown=San%20Francisco
```

About flashVars properties encoding

The values of the `flashVars` properties must be URL encoded. The format of the string is a set of name-value pairs separated by an ampersand (&). You can escape special and nonprintable characters with a percent symbol (%) followed by a two-digit hexadecimal value. You can represent a single blank space by using the plus sign (+).

The encoding for `flashVars` properties is the same as the page. Internet Explorer provides UTF-16-compliant strings on the Windows platform. Netscape sends a UTF-8-encoded string to Flash Player.

Most browsers support a `flashVars` string or query string up to 64 KB (65535 bytes) in length. They can include any number of name-value pairs.

Using the SWF file path to pass request data

You can append the values of properties to the application’s SWF file path in the wrapper. The `swfUrlStr` property identifies the location of the application’s SWF file. It is the first argument in the `swfobject.embedSWF()` method call.

The following example appends query string parameters to the `swfUrlStr` properties in the custom wrapper:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>wrapper/SwfObjectWithFlashVars.html</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "FlashVarTest";
            attributes.name = "FlashVarTest";
            attributes.align = "middle";
            swfobject.embedSWF(
                "FlashVarTest.swf?firstname=Nick&lastname=Danger",
                "flashContent", "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <body>
        <div id="flashContent"/>
   </body>
</html>
```

Variables you define in this manner are accessible in the same way as `flashVars` properties. For more information, see “Accessing the flashVars properties” on page 233.

Accessing the flashVars properties

To access the values of the `flashVars` properties, you use the FlexGlobals object’s `topLevelApplication.parameters` property. This property points to a dynamic object that stores the parameters as name-value pairs. You can access variables on the parameters object by specifying `parameters.`*`variable_name`*.

In your application, you typically assign the values of the run-time properties to local variables. You assign the values of these properties after the Application’s `creationComplete` event is dispatched. Otherwise, the run-time properties might not be available when you try to assign their values to local variables.

The following example defines the `myName` and `myHometown` variables and binds them to the text of Label controls in the `initVars()` method:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/ApplicationParameters.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="initVars()">
     <s:layout>
          <s:VerticalLayout/>
     </s:layout>

       <fx:Script><![CDATA[
          import mx.core.FlexGlobals;

          /* Declare bindable properties in Application scope. */
          [Bindable]
          public var myName:String;
          [Bindable]
          public var myHometown:String;

          /* Assign values to new properties. */
          private function initVars():void {
               myName = FlexGlobals.topLevelApplication.parameters.myName;
               myHometown = FlexGlobals.topLevelApplication.parameters.myHometown;
          }
     ]]></fx:Script>

     <s:HGroup>
          <s:Label text="Name: "/>
          <s:Label text="{myName}" fontWeight="bold"/>
     </s:HGroup>

     <s:HGroup>
           <s:Label text="Hometown: "/>
           <s:Label text="{myHometown}" fontWeight="bold"/>
     </s:HGroup>
</s:Application>
```

When a user requests this application with the `myName` and `myHometown` parameters defined as `flashVars` properties or as query string parameters, Flex displays their values in the Label controls.

To view all the `flashVars` properties, you can iterate over the `FlexGlobals.topLevelApplication.parameters` properties, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/FlashVarTest.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init()">

     <s:layout>
        <s:HorizontalLayout/>
     </s:layout>

     <fx:Script><![CDATA[
         import mx.core.FlexGlobals;
         private function init():void {
              for (var i:String in FlexGlobals.topLevelApplication.parameters) {
                 ta1.text += i + ":" + FlexGlobals.topLevelApplication.parameters[i] + "\n";
              }
          }
      ]]></fx:Script>

      <s:Label text="flashVars"/>
      <s:RichText id="ta1" width="300" height="200"/>
</s:Application>
```

Accessing JavaScript functions

You can call JavaScript functions in the enclosing HTML page from your application. By allowing Flex to communicate with the browser, you can change style settings, invoke remote processes, or perform any other action that you can normally do from within the page's scripts.

You can even pass data from the application to the enclosing HTML page, process it, and then pass it back to the application. You do this by using the ExternalInterface API or the navigateToURL() method. For more information, see "Using the ExternalInterface API to access JavaScript" on page 235 and "Using the navigateToURL() method in Flex" on page 241.

Whenever you communicate with the enclosing page, you must determine if the browser can handle the kinds of actions that you want to perform. Therefore, you should first determine if the browser supports the objects that you want to use. For some general guidelines for determining browser support, see "Using the ExternalInterface API to access JavaScript" on page 235.

Using the ExternalInterface API to access JavaScript

The easiest way to call JavaScript functions from your application is to use the ExternalInterface API. You can use this API to call any JavaScript method on the wrapper, pass parameters, and get a return value. If the method call fails, Flex returns an exception.

The ExternalInterface API encapsulates checks for browser support, so you are not required to do that when using its methods. However, you can check whether the browser supports the interface by using its `available` property. For more information, see "About the ExternalInterface API" on page 227.

The ExternalInterface API consists of a single class, flash.external.ExternalInterface. This class has the call() static method that you use to facilitate the JavaScript to Flash communication. You can also call methods in your application from the wrapper by using the ExternalInterface API's addCallback() method. For more information, see "About the addCallback() method" on page 253.

Calling JavaScript methods from applications

The ExternalInterface API makes it very simple to call methods in the enclosing wrapper. You use the static call() method, which has the following signature:

```
flash.external.ExternalInterface.call(function_name:String[, arg1, ...]):Object;
```

The *function_name* is the name of the function in the HTML page's JavaScript. The arguments are the arguments that you pass to the JavaScript function. You can pass one or more arguments in the traditional way of separating them with commas, or you can pass an object that is deserialized by the browser. The arguments are optional.

The following example `<fx:Script>` block calls the JavaScript `changeDocumentTitle()` function in the enclosing wrapper by using the `call()` method:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/WrapperCaller.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:HorizontalLayout/>
     </s:layout>
  <fx:Script>
     import flash.external.*;

     public function callWrapper():void {
        var s:String;
        if (ExternalInterface.available) {
           var wrapperFunction:String = "changeDocumentTitle";
           s = ExternalInterface.call(wrapperFunction,ti1.text);
        } else {
           s = "Wrapper not available";
        }
        trace(s);
     }
  </fx:Script>

  <mx:Form>
     <mx:FormItem label="New Title:">
        <mx:TextInput id="ti1"/>
     </mx:FormItem>
  </mx:Form>

  <s:Button label="Change Document Title" click="callWrapper()"/>
</s:Application>
```

In your HTML wrapper, you define a function as you would any other JavaScript function. You can return a value, as the following example shows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>Wrapper Being Called</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "WrapperCaller";
            attributes.name = "WrapperCaller";
            attributes.align = "middle";
            swfobject.embedSWF(
                "WrapperCaller.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
     <SCRIPT LANGUAGE="JavaScript">
         function changeDocumentTitle(a) {
             window.document.title=a;
             alert(a);
             return "successful";
         }
     </SCRIPT>
    <body>
        <div id="flashContent"/>
   </body>
</html>
```

Using the ExternalInterface's `call()` method requires that the embedded application have a name in the wrapper; you must set the `attributes.id` and `attributes.name` properties to the name of the SWF file. If you are using the `<object>` and `<embed>` tags in your wrapper, you must define an `id` attribute on the `<object>` tag and a `name` attribute on the `<embed>` tag. Without these, no call to or from your application will succeed. In addition, these properties cannot contain periods or other special characters.

The `call()` method accepts zero or more arguments, which can be ActionScript types. Flex serializes the ActionScript types as JavaScript numbers and strings. If you pass an object, you can access the properties of that deserialized object in the JavaScript, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/DataTypeSender.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:HorizontalLayout/>
     </s:layout>
  <fx:Script><![CDATA[
     import flash.external.*;

     public function callWrapper():void {
        var s:String;
        if (ExternalInterface.available) {
           var o:Object = new Object();
           o.fname = "Nick";
           o.lname = "Danger";
           var wrapperFunction:String = "receiveComplexDataTypes";
           s = ExternalInterface.call(wrapperFunction, o);
        } else {
           s = "Wrapper not available";
        }
        trace(s);
     }
  ]]></fx:Script>

  <s:Button label="Send" click="callWrapper()"/>
</s:Application>
```

Flex only serializes public, nonstatic variables and read-write properties of ActionScript objects. You can pass numbers and strings as properties on objects, simple objects such as primitive types and arrays, or arrays of simple objects.

The JavaScript code can then access properties of the object, as the following example shows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>Data Type Sender Test</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "DataTypeSender";
            attributes.name = "DataTypeSender";
            attributes.align = "middle";
            swfobject.embedSWF(
                "DataTypeSender.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <SCRIPT LANGUAGE="JavaScript">
        function receiveComplexDataTypes(o) {
            alert("Welcome " + o.fname + " " + o.lname + "!");
            return "successful";
        }
    </SCRIPT>
    <body>
        <div id="flashContent"/>
   </body>
</html>
```

You can also embed objects within objects, such as an Array within an Object, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/ComplexDataTypeSender.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
      <s:HorizontalLayout/>
     </s:layout>
  <fx:Script><![CDATA[
     import flash.external.*;

     public function callWrapper():void {
        var s:String;
        if (ExternalInterface.available) {
           var o:Object = new Object();
           o.fname = "Nick";
           o.lname = "Danger";
           o.b = new Array("DdW","E&T","LotR:TS");
           var wrapperFunction:String = "receiveComplexDataTypes";
           s = ExternalInterface.call(wrapperFunction, o);
        } else {
           s = "Wrapper not available";
        }
        trace(s);
     }
  ]]></fx:Script>

  <s:Button label="Send" click="callWrapper()"/>
</s:Application>
```

The code triggers the `receiveComplexDataTypes()` JavaScript function in the following wrapper:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>Complex Data Type Sender Test</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "ComplexDataTypeSender";
            attributes.name = "ComplexDataTypeSender";
            attributes.align = "middle";
            swfobject.embedSWF(
                "ComplexDataTypeSender.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <SCRIPT LANGUAGE="JavaScript">
        function receiveComplexDataTypes(o) {
            // Get value of fname and lname properties.
            var s = ("Welcome " + o.fname + " " + o.lname + "!\n");
            // Iterate over embedded object's properties.
            for (i=0; i<o.b.length; i++) {
                s +=  i + ": " + o.b[i] + "\n";
            }
            alert(s);
        }
    </SCRIPT>
    <body>
        <div id="flashContent"/>
   </body>
</html>
```

Flex and Flash Player have strict security in place to prevent cross-site scripting. By default, you cannot call script on an HTML page if the HTML page is not in the same domain as the application. However, you can expand the sources from which scripts can be called. For more information, see “About ExternalInterface API security in Flex” on page 252.

You cannot pass objects or arrays that contain circular references. For example, you cannot pass the following object:

```
var obj = new Object();
obj.prop = obj; // Circular reference.
```

Circular references cause infinite loops in both ActionScript and JavaScript.

Using the navigateToURL() method in Flex

The navigateToURL() method loads a document from a specific URL into a window or passes variables to another application at a defined URL. You can use this method to call JavaScript functions in the HTML page that encloses an application.

You should not confuse the functionality of the `navigateToURL()` method with the `load()` method of the URLLoader class. The URLLoader class loads a specified URL into an object for manipulation with ActionScript. The `navigateToURL()` method navigates to the specified URL with a browser.

In most cases, you should use the ExternalInterface API to perform Flex-to-wrapper communication. However, the `navigateToURL()` method is not part of the ExternalInterface API and, therefore, does not have as stringent a set of requirements for which browsers support it. These requirements are described in “About the ExternalInterface API” on page 227.

The `navigateToURL()` method is restricted by the security sandbox in which the SWF file is running. Its use relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` parameters define. You set the values of the `allowScriptAccess` and `allowNetworking` parameters in the SWF file’s wrapper.

For more information on these parameters, see “Creating a wrapper” on page 2552. For more information on security restrictions, see “Security” on page 117.

The navigateToURL() method syntax

The navigateToURL() method is in the flash.net package. It has the following signature:

```
navigateToURL(request:URLRequest, window:String):void
```

The *request* argument is a URLRequest object that specifies the destination. The *window* argument specifies whether to launch a new browser window or load the new URL into the current window. The following table describes the valid values for the *window* argument:

Value	Description
`_self`	Specifies the current frame in the current window.
`_blank`	Specifies a new window. This new window acts as a pop-up window in the client's browser, so you must be aware that a pop-up blocker could prevent it from loading.
`_parent`	Specifies the parent of the current frame.
`_top`	Specifies the top-level frame in the current window.

You pass a URLRequest object to the `navigateToURL()` method. This object defines the URL target, variables, method (POST or GET), window, and headers for the request. The following example defines a simple URL to navigate to:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/SimplestNavigateToURL.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
      <s:HorizontalLayout/>
     </s:layout>
  <fx:Script><![CDATA[
     import flash.net.*;
     public function openNewWindow(event:MouseEvent):void {
        var u:URLRequest = new URLRequest("http://www.adobe.com/flex");
        navigateToURL(u,"_blank");
     }
  ]]></fx:Script>
  <s:Button label="Open New Window" click="openNewWindow(event)"/>
</s:Application>
```

The `navigateToURL()` method URL encodes the value of the `url` argument.

To send data with a URLRequest object, you can append variables to the request string. The following example launches a new window and passes a search term to the URL:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/SimpleNavigateToURL.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import flash.net.*;
            public function executeSearch(event:MouseEvent):void {
                var u:URLRequest = new URLRequest("http://www.google.com/search?hl=en&q=" +
ta1.text);
                navigateToURL(u,"_blank");
            }
        ]]>
    </fx:Script>
    <s:TextArea id="ta1"/>
    <s:Button label="Search" click="executeSearch(event)"/>
</s:Application>
```

In addition to appending strings, you can also use the `data` property of the URLRequest to add URL variables to a GET or POST request. The query string parameters are of type URLVariables. Flex adds ampersand delimiters for you in the URL request string. The following example adds `name=fred` to the URLRequest:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/NavigateWithGetMethod.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
      <s:HorizontalLayout/>
     </s:layout>
  <fx:Script><![CDATA[
     import flash.net.*;
     public function openNewWindow(event:MouseEvent):void {
        var url:URLRequest = new
URLRequest("http://aspexamples.adobe.com/chart_examples/stocks.aspx");
        var uv:URLVariables = new URLVariables();
        url.method = "GET";
        uv.tickerSymbol = ti1.text;
        url.data = uv;
        navigateToURL(url,"_blank");
     }
  ]]></fx:Script>
  <mx:TextInput id="ti1" text="STRK"/>

  <s:Button label="Open New Window" click="openNewWindow(event)"/>
</s:Application>
```

To use POST data with the URLRequest object, set the value of the URLRequest object’s `method` property to POST.

Opening multiple windows with the navigateToURL() method

You can open any number of new browser windows with calls to the `navigateToURL()` method. However, because ActionScript is asynchronous, if you tried a simple loop over the method, Flash Player would only open the browser window for the last call. To avoid this, you use the `callLater()` method. This method instructs Flash Player to open a new browser window on each new frame in your application. The following example uses the `callLater()` method to open multiple browser windows with the `navigateToURL()` method:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/NavigateToMultipleURLs.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
      <s:HorizontalLayout/>
     </s:layout>
  <fx:Script><![CDATA[
     import flash.net.navigateToURL;
     private var listingData:Array=[
        "http://flex.org",
        "http://www.adobe.com",
        "http://aspexamples.adobe.com"
     ];
     private function openWindows(n: Number):void {
        if (n < listingData.length) {
           navigateToURL(new URLRequest(listingData[n]), '_blank');
           callLater(callLater, [openWindows, [n+1]]);
        }
     }
  ]]></fx:Script>
  <s:Button label="Click Me to Open Multiple Windows"
    click="openWindows(0)"/>
</s:Application>
```

Some browsers block popups. You might need to disable popup blocking to view this example. In some cases, you can also Control+click the button to avoid the popup blocker behavior.

Calling JavaScript functions with the URLRequest object

You can use theURLRequest to call a JavaScript function by embedding that function in the first parameter of the method, as the following example shows:

```
public var u:URLRequest = new URLRequest("javascript:window.close()");
```

The previous code does not work on all browsers. You should include code that detects the browser type, and closes the window based on the type of browser. Also, some browsers, such as Internet Explorer, behave in unexpected ways when you invoke a URLRequest that contains JavaScript. Here are some examples:

- Executes the JavaScript URLs asynchronously. This means that it is possible to have multiple calls to the navigateToURL() method that is trying to execute JavaScript methods, and have only the last one occur. Each one overwrites the next.
- Stops all other navigation, such as loading of images and IFRAMEs when you call a JavaScript URL.
- Displays a security warning dialog box if the URL contains ampersand (&) or question mark (?) characters.

Invoking JavaScript with the navigateToURL() method

You can use the `navigateToURL()` method to invoke JavaScript functions on the HTML page in which the application runs. However, when passing parameters by using the `navigateToURL()` method, you must enclose each parameter in quotation marks, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/CallingJavaScriptWithNavigateToURL.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
      <s:HorizontalLayout/>
     </s:layout>
    <fx:Script><![CDATA[
        import flash.net.*;
        public function callWrapper(e:Event):void {
            var eType:String = String(e.type);
            var eName:String = String(e.currentTarget.id);
          var u:URLRequest = new URLRequest("javascript:catchClick('" + eName + "','" + eType
+ "')");
            navigateToURL(u,"_self");
        }
    ]]></fx:Script>
    <s:Button id="b1" click="callWrapper(event)" label="Call Wrapper"/>
</s:Application>
```

The enclosing HTML wrapper includes the JavaScript to process this call, as the following example shows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>navigateToURL() Wrapper</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "CallingJavaScriptWithNavigateToURL";
            attributes.name = "CallingJavaScriptWithNavigateToURL";
            attributes.align = "middle";
            swfobject.embedSWF(
                "CallingJavaScriptWithNavigateToURL.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <SCRIPT LANGUAGE="JavaScript">
        function catchClick(name, type) {
            alert(name + " triggered the " + type + " event.");
        }
    </SCRIPT>
    <body>
        <div id="flashContent"/>
   </body>
</html>
```

You can also use the `navigateToURL()` method with basic JavaScript functions in the URLRequest itself. For example, you can launch the default e-mail client with a single line of code:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- wrapper/EmailWithNavigateToURL.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:HorizontalLayout/>
     </s:layout>
  <fx:Script>
     import flash.net.*;
     public function sendMail(e:Event):void {
        var u:URLRequest = new URLRequest("mailto:" + ti1.text);
        navigateToURL(u,"_self");
     }
  </fx:Script>
  <s:Button id="b1" click="sendMail(event)" label="Send Mail"/>
  <mx:Form>
     <mx:FormItem>
        <mx:Label text="Email Address: "/>
     </mx:FormItem>
     <mx:FormItem>
        <mx:TextInput id="ti1"/>
     </mx:FormItem>
  </mx:Form>
</s:Application>
```

Not all browsers support invoking the `javascript` protocol with the `navigateToURL()` method. If possible, you should use the call() method of the ExternalInterface API to invoke JavaScript methods in the enclosing HTML page. For more information, see "Using the ExternalInterface API to access JavaScript" on page 235.

Accessing Flex from JavaScript

You can call Flex methods from your enclosing wrapper by using the ExternalInterface API. You do this by adding a public method in your application to a list of callable methods. In your application, you add a local Flex function to the list by using the addCallback() method of the ExternalInterface API. This method registers an ActionScript method as callable from the JavaScript or VBScript in the wrapper.

Note: *This feature requires that the client is running certain browsers. For more information, see "About the ExternalInterface API" on page 227.*

The signature for the `addCallback()` method is as follows:

```
addCallback(function_name:String, closure:Function):void
```

The *function_name* parameter is the name by which you call the Flex function from your HTML page's scripts. The *closure* parameter is the local name of the function that you want to call. This parameter can be a method on the application or an object instance.

The following example declares the `myFunc()` function to be callable by the wrapper:

```
<?xml version="1.0"?>
<!-- wrapper/AddCallbackExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="initApp()">
    <fx:Script>
        import flash.external.*;
         public function initApp():void {
            ExternalInterface.addCallback("myFlexFunction",myFunc);
         }
         public function myFunc(s:String):void {
            l1.text = s;
         }
    </fx:Script>
    <s:Label id="l1"/>

</s:Application>
```

To call the Flex method from the HTML wrapper, you get a reference to the movie object. The name of the movie is the same value as the `attributes.id` and `attributes.name` properties. In this case, it is AddCallbackExample, as the following example wrapper shows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>addCallback() Wrapper</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "AddCallbackExample";
            attributes.name = "AddCallbackExample";
            attributes.align = "middle";
            swfobject.embedSWF(
                "AddCallbackExample.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <SCRIPT LANGUAGE="JavaScript">
        function callApp() {
            window.document.title = document.getElementById("newTitle").value;
            var AddCallbackExample = document.getElementById("AddCallbackExample");
            AddCallbackExample.myFlexFunction(window.document.title);
        }
    </SCRIPT>
    <body>
        <form id="f1">
            Enter a new title: <input type="text" size="30" id="newTitle"
onchange="callApp()">
        </form>
        <div id="flashContent"/>
   </body>
</html>
```

If your wrapper uses `<object>` and `<embed>` tags, the movie object is the same value as the `id` and `name` properties of the `<object>` and `<embed>` tags. You then call the method on that object, passing whatever parameters you want, as the following example shows:

```
<html>
<head>
<title>wrapper/AddCallbackWrapper.html</title>
</head>
<body scroll='no'>
<SCRIPT LANGUAGE="JavaScript">
    function callApp() {
        window.document.title = document.getElementById("newTitle").value;
        mySwf.myFlexFunction(window.document.title);
    }
</SCRIPT>
<h1>AddCallback Wrapper</h1>
<form id="f1">
    Enter a new title: <input type="text" size="30" id="newTitle" onchange="callApp()">
</form>
<table width='100%' height='100%' cellspacing='0' cellpadding='0'>
    <tr>
        <td valign='top'>
         <object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' height='200'
width='400'>
                <param name='src' value='AddCallbackExample.swf'/>
                <param name='flashVars' value=''/>
                <embed name='mySwf' src='AddCallbackExample.swf' height='100%' width='100%'
flashVars=''/>
            </object>
        </td>
    </tr>
</table>
</body></html>
```

If there is no method with the appropriate name in the application or that method hasn't been made callable, the browser throws a JavaScript error.

Getting a reference to the application object and other DOM objects is not the same in Internet Explorer as it is in Chrome or FireFox. If you do not know which browser your users will be using when they request your application, you should make your wrapper's script browser independent. For some ideas on detecting the browser type, you can look at the code in the swfobject.js file. To cover most cases, you typically use the `getElementById()` method.

Flash Player has strict security in place to prevent cross-site scripting. By default, Flex methods are not callable by HTML scripts. You must explicitly identify them as callable. You also cannot call a Flex method from an HTML page if the HTML page is not in the same domain as the application. However, it is possible to expand the sources from which Flex methods are called. For more information, see "About the addCallback() method" on page 253.

Handling browsers that disable JavaScript

In some cases, the client's browser either does not support JavaScript or the user has purposely disabled it. You can use the `<noscript>` tag in the wrapper to define what happens when this user tries to run your applications. Most commonly, you use the `<object>` and `<embed>` tags inside the wrapper's `<noscript>` block to embed the application. However, without scripting, deep linking and Express Install functionality is not available.

The following is an example of a `<noscript>` block that embeds an application built with Flex:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>wrapper/NoScriptWrapper.html</title>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    </head>
    <body>
       <noscript>
            <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" width="100%"
height="100%" id="AddCallbackExample">
                <param name="movie" value="TestProject.swf" />
                <param name="quality" value="high" />
                <param name="bgcolor" value="#ffffff" />
                <param name="allowScriptAccess" value="sameDomain" />
                <!--[if !IE]>
            <object type="application/x-shockwave-flash" data="TestProject.swf" width="100%"
height="100%">
                    <param name="quality" value="high" />
                    <param name="bgcolor" value="#ffffff" />
                    <param name="allowScriptAccess" value="sameDomain" />
                <![endif]-->
                <!--[if gte IE 6]>
                <p>
                Either scripts and active content are not permitted to run or Adobe Flash 
Player version
                10.0.0 or greater is not installed.
                </p>
                <![endif]-->
                    <a href="http://www.adobe.com/go/getflashplayer">
                         <img 
src="http://www.adobe.com/images/shared/download_buttons/get_flash_player.gif" alt="Get Adobe 
Flash Player" />
                    </a>
                <!--[if !IE]>
                </object>
                <![endif]-->
            </object>
        </noscript>
   </body>
</html>
```

If your application *requires* functionality that is available only to scripting-enabled browsers (such as deep linking or access to the ExternalInterface API), then you can insert a message in the `<noscript>` block that warns the user against accessing your application while JavaScript is disabled. The following example warns users when someone with JavaScript disabled tries to run your application:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>wrapper/NoScriptWarning.html</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "TestProject";
            attributes.name = "TestProject";
            attributes.align = "middle";
            swfobject.embedSWF(
                "TestProject.swf",
                "flashContent", "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <body>
        <div id="flashContent"/>
       <noscript>
            <EM>Your browser either does not support JavaScript or has disabled it.
            Some features of this application require JavaScript.
            Please enable JavaScript and reload this page.</EM>
       </noscript>
   </body>
</html>
```

About ExternalInterface API security in Flex

Allowing applications built with Flex to call embedded scripts on HTML pages and vice versa is subject to stringent security constraints. By default, scripts on the HTML page can communicate only with ActionScript in an application if the page and the application are in the same domain. You can expand this restriction to include applications outside of the domain.

About the call() method

The success of the call() method depends on the HTML page's use of the `allowScriptAccess` parameter. This parameter is not an ActionScript mechanism; it is an HTML parameter. Its value determines whether your application can call JavaScript in the HTML page, and it applies to all functions on the page. The default value of `allowScriptAccess` only allows communication if the application and the HTML page are in the same domain.

In an HTML wrapper that uses SWFObject 2, you set the value of the `allowscriptaccess` property on the params object that is passed to the `swfobject.embedSWF()` method. The following example sets the value of the `allowScriptAccess` property to `sameDomain`:

```
var params = {};
params.quality = "high";
params.bgcolor = "#ffffff";
params.allowscriptaccess = "sameDomain";
```

The following table describes the valid values of the `allowScriptAccess` parameter:

Value	**Description**
`never`	The `call()` method fails.
`sameDomain`	The `call()` method succeeds if the calling application is from same domain as the HTML page that loaded the application. This is the default value.
`always`	The `call()` method succeeds, regardless of whether the calling application is in the same domain as the HTML page that loaded the application.

If your HTML wrapper uses the `<object>` and `<embed>` tags, you set the value of the `allowScriptAccess` property on the `<object>` tag as follows:

```
<object id='SendComplexDataTypes' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
allowScriptAccess='sameDomain' height='100%' width='100%'>
```

On the `<embed>` tag, set the property as follows:

```
<embed name='SendComplexDataTypes.mxml.swf' src='SendComplexDataTypes.mxml.swf'
allowScriptAccess='sameDomain' height='100%' width='100%' flashvars=''/>
```

About the addCallback() method

Flex prevents JavaScript methods from calling just any method in your application by requiring that you explicitly make the method callable. The default for all methods is to not be callable from JavaScript. The ExternalInterface API enables a SWF file to expose a specific interface that JavaScript can call.

By default, an HTML page can only communicate with the ActionScript in your application if it originates from the same domain. You allow HTML pages outside of the application's domain to call methods of your application using the allowDomain() method. For more information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Runtime Shared Libraries

Introduction to RSLs

One way to reduce the size of your applications' SWF files is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets can be loaded and used by any number of applications at run time, but are transferred only once to the client. These shared files are known as *Runtime Shared Libraries* or *RSLs*.

When multiple applications share a core set of components or classes, clients can download those assets only once as an RSL rather than once for each application. The RSLs are persisted on the client disk so that they do not need to be transferred across the network a second time. The resulting file size for the applications can be reduced. The benefits increase as the number of applications that use the RSL increases.

Applications built with Flex support the following types of RSLs:

- Framework RSLs — Libraries of components and framework classes that all applications can share. Framework RSLs are precompiled for you. Adobe provides hosted, signed framework RSLs that you can link to from any application that has internet access. For more information, see "Using the framework RSLs" on page 260.
- Standard RSLs — A library of custom classes created by you to use across applications that are in the same domain. Standard RSLs are stored in the browser's cache. For more information, see "About standard RSLs" on page 267.
- Cross-domain RSLs — A library of custom classes, like standard RSLs, with the difference being that they can be loaded by applications in different domains and sub-domains. Cross-domain RSLs are stored in the browser's cache. For more information, see "About cross-domain RSLs" on page 268.

You can create your own RSLs from custom libraries. You do this by using either the Adobe® Flex® Builder's™ Build Project option for your Flex Library Project or the compc command-line compiler.

About linking

Understanding library linking can help you understand how RSLs work and how you can most benefit from their use. The Flex compilers support static linking and dynamic linking for class libraries (including RSLs). Static linking is the most common type of linking when compiling an application. However, dynamic linking lets you take advantage of RSLs to achieve a reduction of the final SWF file size and, therefore, a reduction in the application download time.

When you use *static linking*, the compiler includes all referenced classes and their dependencies in the application SWF file. The end result is a larger file that takes longer to download than a dynamically-linked application, but loads and runs quickly because all the code is in the SWF file.

To statically link a library's definitions into your application, you use the `library-path` and `include-libraries` compiler options to specify locations of SWC files.

When you use the `library-path` option, the compiler includes only those classes required at compile time in the SWF file, so the entire contents of a library are not necessarily compiled into the SWF file. The `include-libraries` option includes the entire contents of the library, regardless of which classes are required. You can also use the `source-path` and `includes` options to embed individual classes in your SWF file.

In Flash Builder, you use the Project > Properties > Flex Build Path > Library Path dialog to add libraries to your project. To statically link a library at compile time, you select Merged Into Code for the library's Link Type. This includes only those classes that are used in the application, so it is the equivalent of the `library-path` compiler option.

If you statically link any part of a library into your application, you cannot use that library as an RSL.

Dynamic linking is when some classes used by an application are left in an external file that is loaded at run time. The result is a smaller SWF file size for the main application, but the application relies on external files that are loaded at run time. Dynamic linking is used by modules, runtime stylesheets, and RSLs.

When you want to use a dynamically-linked library, you instruct the compiler to exclude that library's contents from the application SWF file when you compile the application. You must provide link-checking at compile time even though the classes are not going to be included in the final SWF file. At run time, the application loads the entire library into the application SWF file, which can result in slower startup times and greater memory usage.

You can use the `runtime-shared-library-path` (Flex 4 and later) and `runtime-shared-libraries` (Flex 3) options to specify the location of dynamically-linked libraries.

You can also use the `external-library-path`, `externs`, or `load-externs` compiler options to specify the files to dynamically link into an application. These options instruct the compiler to exclude classes and libraries from the application, but to check links against them and prepare to load them at run time. The `external-library-path` option specifies SWC files or directories for dynamic linking. The `externs` option specifies individual classes or symbols for dynamic linking. The `load-externs` option specifies an XML file that describes what classes to use for dynamic linking. These options are most often used when externalizing assets from modules so that the module and the application do not contain overlapping class definitions. The `runtime-shared-library-path` option provides all the arguments to use external libraries as RSLs.

In Flash Builder, to use dynamically-linked libraries, you specify either RSL or External as the Link Type in the Library Path dialog for the library.

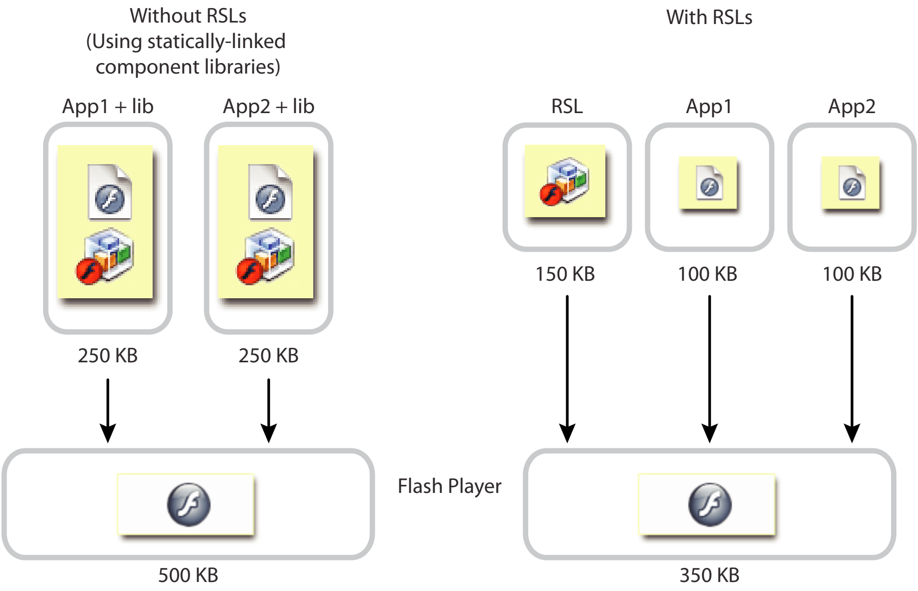
You can view the linking information for your application by using the `link-report` compiler option. This generates a report that has the same syntax as the file that you load with the `load-externs` option, so you can use it as an argument to that option without changing it. For more information about this report, see “Examining linker dependencies” on page 2308.

For more general information about the command-line compiler options, see “Flex compilers” on page 2164.

RSL benefits

The following example shows the possible benefit of separating shared components into an RSL. In this example, the library’s size is 150 KB (kilobytes) and the compiled application’s size is 100 KB. Without RSLs, you merge the library into both applications for an aggregate download size of 500 KB. If you add a third or fourth application, the aggregate download size increases by 250 KB for each additional application.

With RSLs, the RSL needs to be downloaded once only. For two applications that use the same RSL, the result is an aggregate download size of 350 KB, or a 30% reduction. If you add a third or fourth application, the aggregate download size increases by 100 KB instead of 250KB for each additional application. In this example, the benefits of using an RSL increase with each new application.



In this example, the applications with statically-linked libraries run only after Adobe® Flash® Player loads the 250 KB for each application. With dynamically linked RSLs, however, only the first application must load the entire 250 KB (the combined size of the application and the RSL). The second application runs when just 100 KB loads because the RSL is cached.

The illustrated scenario shows one possible outcome. If your applications do not use all of the components in the RSL, the size difference (and, as a result, the savings in download time) might not be as great. Suppose that each application only uses half of the components in the RSL. If you statically link the library, only those classes that are used are included; the output, as a result, is 100 KB + 75 KB for the first application and the library and 100 KB + 75 KB for the second application and the library, or an aggregate download size of 350 KB. When you use a library as an RSL, its entire SWF file must be transferred across the network and loaded by the application at run time, regardless of how much of that library is actually used. In this second case, the combined download size when using RSLs and when not using RSLs is the same.

In general, the more applications that use a common RSL, the greater the benefit.

RSL considerations

RSLs are not necessarily beneficial for all applications. You should try to test both the download time and startup time of your application with and without RSLs.

Standard RSLs can not be shared across domains. If a client runs an application in domain1.com and uses an RSL, and then launches an application in domain2.com that uses the same RSL, the client downloads the RSL twice even though the RSL is the same. You can overcome this limitation of standard RSLs by using cross-domain RSLs.

Cross-domain RSLs can be loaded by any application, even if that application is not in the same domain. They do, however, require that you create and check a digest when the RSL is loaded. This can increase startup time of the application by a small amount.

Framework RSLs can also be loaded by any application. To take advantage of the fact that framework RSLs can be cached in the Player cache, the client must be running a recent version of Flash Player. Not all clients necessarily have the latest Player, so loading a framework RSL might fail. In these cases, you can specify a failover RSL.

An RSL usually increases the startup time of an application. This is because the entire library is loaded into an application regardless of how much of the RSL is actually used. For this reason, make your RSLs as small as possible. This contrasts with how statically-linked libraries are used. When you compile an application, the compiler extracts just the components it needs from those component libraries.

If you have several applications that share several libraries, it might be tempting to merge the libraries into a single library that you use as an RSL. However, if the individual applications generally do not use more than one or two libraries each, the penalty for having to load a single, large RSL might be higher than it would be to have the applications load multiple smaller RSLs.

In this case, test your application with both a single large RSL and multiple smaller RSLs, because the gains are largely application specific. It might be better to build one RSL that has some extra classes than to build two RSLs, if most users will load both of them anyway.

If you have overlapping classes in multiple RSLs, be sure to synchronize the versions so that the wrong class is never loaded.

You cannot use RSLs in ActionScript-only projects if the base class is Sprite or MovieClip. RSLs require that the application's base class, such as Application or SimpleApplication, understand RSL loading.

In general, you should disable RSLs when compiling modules, CSS SWF files, and applications that will be loaded as child SWFs in a parent application. The parent application will likely use RSLs and dynamically link the required classes.

About caching

RSLs are cached when they are first used. When they are needed by another application, they can be loaded from the cache rather than across the network. Caching is one of the benefits of RSLs, because disk access is much faster than network access.

The type of caching used by an RSL is based on the type of RSL. Standard or cross-domain RSLs are stored in the browser's cache. If the user clears their cache, the RSLs are removed and must be downloaded again the next time they are needed. Unsigned framework RSLs are also stored in the browser's cache.

Signed framework RSLs are stored in the Player cache. This is a special cache that is maintained by Flash Player. To clear this cache, clients must invoke the Settings Manager. RSLs stored in this cache are signed and therefore can be used by any application without the need for a cross-domain policy file.

For more information about the framework cache, see "About the Player cache" on page 263.

Common RSL tasks

The techniques in this section apply to framework and custom RSLs.

Removing unused RSLs

By default, the compiler removes unused RSLs when an application is compiled. This prevents an application from having to download RSLs that it does not use. You can override this behavior by preventing the compiler from removing unused RSLs.

To prevent unused RSLs from being removed, set the `remove-unused-rsls` compiler option to `false`. The default value is `true`.

You can configure the `remove-unused-rsls` compiler option in the flex-config.xml file, as the following example shows:

```
<remove-unused-rsls>true</remove-unused-rsls>
```

In Flash Builder, to toggle the `remove-unused-rsls` option:

1 Select Project > Properties.

2 Select Flex Build Path. The Flex Build Path dialog appears.

3 Select the Library Path panel.

4 Select or deselect the "Remove unused RSLs" checkbox.

5 Click OK to save your changes.

Forcing RSLs to load

You can force the compiler to load particular RSLs when compiling your application. You do this with the `force-rsls` compiler option.

Forcing RSLs to load can be useful if you use classes that are only referenced indirectly (sometimes referred to as "soft references"), so the compiler might not know they are needed.

When forcing RSLs to load, you must configure them in the `runtime-shared-library-path` option in addition to specifying them with the `force-rsls` option.

The following example forces the osmf.swc and rpc.swc RSLs to be loaded:

```
mxmlc -force-rsls=libs/osmf.swc,libs/rpc.swc MyApp.mxml
```

You can configure the `force-rsls` compiler option in the flex-config.xml file, as the following example shows:

```
<runtime-shared-library-settings>
    <force-rsls>
        <path-element>libs/osmf.swc</path-element>
        <path-element>libs/rpc.swc</path-element>
    </force-rsls>
</runtime-shared-library-settings>
```

In Flash Builder, to force an RSL to load:

1. Select Project > Properties.
2. Select Flex Build Path. The Flex Build Path dialog appears.
3. Select the Library Path panel.
4. Select the RSL from the “Build path libraries” list and click Edit. The Library Path Item Options dialog box appears.
5. Select the “Force load RSL” checkbox.
6. Click OK to save your changes.

Viewing required RSLs

By default the compiler outputs a list of RSLs that an application uses. This is viewable in the command line compiler’s output.

The following is an example of the output from an application that loads the framework, textLayout, spark, sparkskins, and osmf RSLs:

```
Required RSLs:
    http://fpdownload.adobe.com/pub/swz/flex/4.6.0/framework_4.6.0.swf with 1 failover.
    http://fpdownload.adobe.com/pub/swz/flex/4.6.0/textLayout_2.0.0.139.swf with 1 failover.
    http://fpdownload.adobe.com/pub/swz/flex/4.6.0/spark_4.6.0.swf with 1 failover.
    http://fpdownload.adobe.com/pub/swz/flex/4.6.0/osmf_1.0.0.16316.swf with 1 failover.
```

The list of required RSLs displays the expected location of the RSLs at runtime. In this case, the expected location is the signed SWFs available on Adobe’s website. The output also notes that there is a failover location in case the Adobe RSLs are unavailable. You can see this location in the flex-config.xml file.

Note that if `static-link-runtime-shared-libraries` option is set to `true`, then no RSLs are listed because they are not used. Static linking is used instead.

Disabling RSLs

You can disable RSLs when you compile your application by setting the `static-link-runtime-shared-libraries` compiler option to `true`.

In general, disabling RSLs is useful only when compiling style SWF files, resource bundles, or other non-application assets.

Using RSLs with modules and sub-applications

RSLs are designed to work efficiently with modules and sub-applications. When using RSLs with modules and sub-applications, note that:

- Main applications only load RSLs that are needed, but create placeholders for the remaining framework RSLs. This lets the module or sub-application load other framework RSLs into the main application when they need it.
- Sub-applications and modules share framework RSLs by default.
- Sub-applications and modules can share custom RSLs if you configure the application domain that the RSLs are loaded into.

Using placeholder RSLs

When you compile an application with Flex, the default behavior is to compile the application against the framework (or default) RSLs. When the application runs, it loads only the framework RSLs that it actually uses, and creates placeholders for the remaining framework RSLs. When the application loads a module or sub-application that requires one of the framework RSLs for which there is a placeholder, the module or sub-application then loads that RSL into the main application.

If a module or sub-application requires a framework RSL that is already loaded, it will not load the RSL again. It will instead use the existing RSL.

To ensure that placeholders for framework RSLs are used, set the `remove-unused-rsls` compiler argument is `true` when compiling the main application, sub-applications, and modules. This is the default.

By default, framework RSLs are loaded into the top-most application domain that contains a placeholder for that RSL. As a result, modules or sub-applications can share RSLs that were loaded by other modules or sub-applications, as long as you do not restrict the domain into which the RSL was loaded.

Specifying domains for RSLs (advanced)

When loading custom or framework RSLs into a sub-application or module, you can specify the application domain into which the RSL is loaded. You do this with the `application-domain` compiler argument. This lets you restrict RSLs from being shared, or ensure that they are shared, among modules and sub-applications.

The `application-domain` compiler argument has the following syntax:

```
runtime-shared-library-settings.application-domain=
    path-element,application-domain-target
```

The `path-element` option specifies the location of the RSL’s SWC library file. This RSL must also be defined by the `runtime-shared-library-path` option.

The `application-domain-target` option takes one the following values:

- `default` — Loads the RSL into the top-most application domain with a placeholder. If no placeholder is found, then the RSL is loaded into the current application domain.
- `current` — Loads the RSL into the module or sub-application’s current application domain. Modules and sub-applications will not be able to share this RSL with anything other than child modules or sub-applications.
- `top-level` — Loads the RSL into the application domain of the top-level SystemManager. This is the main or root application. In this case, all modules and sub-applications will be able to share this RSL.
- `parent` — Loads the RSL into the application domain of the parent application or module.

The following example snippet from the flex-config.xml file defines a custom RSL, MyLibrary, and instructs it to be loaded into the top-level application domain:

```
<runtime-shared-library-path>
    <path-element>libs/MyLibrary.swc</path-element>
    <rsl-url>bin/MyLibrary.swf</rsl-url>
    <policy-file-url></policy-file-url>
</runtime-shared-library-path>
<runtime-shared-library-settings>
    <application-domain>
        <path-element>libs/MyLibrary.swc</path-element>
        <application-domain-target>top-level</application-domain-target>
    </application-domain>
</runtime-shared-library-settings>
```

Note that RSLs can only be shared between sub-applications and modules that have a common parent. A sub-application or module cannot share RSLs with a sub-application that is untrusted (sandboxed) or loaded for version compatibility (multi-versioned).

Changing RSL application domains in Flash Builder

To change the application domain of an RSL in Flash Builder:

1. Select Project > Properties.
2. Select Flex Build Path. The Flex Build Path dialog appears.
3. Select the Library Path panel.
4. Select the RSL from the "Build path libraries" list and click Edit. The Library Path Item Options dialog appears.
5. Select the appropriate application domain from the Application Domain drop-down box.
6. Click OK to save your settings.

Preventing framework RSLs from being shared among modules and sub-applications

For framework RSLs, the default is to load an RSL into the highest application domain that has a placeholder for the RSL (typically the main application). The result is that once a module or sub-application loads an RSL, all subsequent modules or sub-applications that need this RSL will not have to load it. The downside to this is that when the module or sub-application is unloaded, the RSL is not unloaded.

To prevent a framework RSL from being loaded into the main application, you can specify `"current"` for the application domain. This loads the RSL into the module or sub-application's application domain. When the module or sub-application is unloaded, so is the RSL. Other modules or sub-applications will not have access to it. You should not specify `"current"` for the application domain when loading the OSMF or MX RSLs. These RSLs are always loaded into the root application.

Sharing custom RSLs among modules and sub-applications

For custom RSLs, the default is to load the RSL into the module or sub-application's current application domain. The result is that the module or sub-application does not share the RSL with sibling modules or sub-applications, but only with child sub-applications or modules that it loads.

To load a custom RSL into an application domain so that it can then be shared with other modules and sub-applications, you specify `"top-level"` for the application domain. This loads the custom RSL into the main application. All modules and sub-applications will then be able to access that RSL.

You should ensure that applications or modules do not statically link classes in a custom RSL. If those classes are loaded before the RSL is loaded, then class conflicts can occur when the RSL is loaded. One technique to use when doing this is to ensure that all modules, sub-applications, and the main application share the same RSL definitions. If this is the case, classes will not be statically linked when they exist in an RSL.

Using the framework RSLs

Every application built with Flex uses some aspects of the Flex framework, which is a relatively large set of classes that define the infrastructure of an application. If a client uses two different applications, the applications will likely load overlapping class definitions. The result is that applications are larger and load slower than they should. This can be a problem for users who are on dialup or slow network connections. It also leads to the perception that applications load more slowly than other web-based applications.

To overcome these limitations, Flex provides framework RSLs. These libraries are comprised of the Flex class libraries and can be used with any application built with Flex. The framework RSLs are precompiled libraries of framework classes and components.

Framework RSLs come in two versions: signed and unsigned.

- *Signed framework RSLs* are cached in a special Player cache and in the browser cache. They can be accessed by any application regardless of that application's originating domain. They only need to be downloaded to the client once. When the browser's cache is cleared, signed framework RSLs persist in the special Player cache.
- *Unsigned framework RSLs* are cached in the browser cache and can only be used by applications that have access to the RSL's source domain. When the browser's cache is cleared, unsigned framework RSLs are removed just like any other file in the cache.

By default, framework RSLs are dynamically linked into your applications. When an application loads on the client, the client attempts to load the framework RSLs if they are not already in the client's cache. Typically, they attempt to load the signed framework RSL that is hosted by Adobe first. The location and order of RSLs loaded is defined in the flex-config.xml file. You can override these values on the command line or by using the Library Path panel of the Flex Build Path properties panel in Flash Builder.

Flash Player 9.0.115 and later support loading signed framework RSLs. These RSLs can be loaded by applications in different domains. The framework RSLs are signed and have the extension SWZ.

Only Adobe can create signed RSLs, and only signed RSLs can be stored in the Player cache. If you create an RSL that contains a custom library, it will be unsigned. You cannot sign it. If a Player with a version earlier than 9.0.115 attempts to load a framework RSL, then Flash Player skips it and loads a failover RSL, if one was specified when the application was compiled. (Flex 4 applications require Player 10.0 or later, so this might not be an issue for you.)

Only applications compiled with the Flex 3 and later compilers can use signed framework RSLs. Applications compiled with earlier versions of the compilers cannot use signed framework RSLs.

Included framework RSLs

If your application uses framework RSLs, the client attempts to load them from Adobe first. If the Adobe site is unavailable, or the client's network access is restricted, you can instruct the client to use the included framework RSLs instead.

The framework RSLs that are included in the SDK installation are located in the *flex_sdk_dir*/frameworks/rsls directory. The naming convention includes the version number of Flex, plus the build number of the compiler that you currently use. The following framework RSLs are included:

File Name	Description
advancedgrids_*version.build*.swf advancedgrids_*version.build*.swz	Signed and unsigned RSLs containing AdvancedDataGrid and OLAPDataGrid classes. Flash Builder only.
charts_*version.build*.swf charts_*version.build*.swz	Signed and unsigned RSLs containing charting classes.
framework_*version.build*.swf framework_*version.build*.swz	Signed and unsigned RSLs containing framework classes.
mx_*version.build*.swf mx_*version.build*.swz	Signed and unsigned RSLs containing MX components and supporting classes.

File Name	Description
osmf_*version.build*.swf osmf_*version.build*.swz	Signed and unsigned RSLs containing OSMF-related classes.
rpc_*version.build*.swf rpc_*version.build*.swz	Signed and unsigned RSLs containing data services classes.
spark_*version.build*.swf spark_*version.build*.swz	Signed and unsigned RSLs containing Spark components and supporting classes.
spark_dmv_*version.build*.swf spark_dmv_*version.build*.swz	Signed and unsigned RSLs containing Spark classes used by the AdvancedDataGrid and OLAPDataGrid classes.
sparkskins_*version.build*.swf sparkskins_*version.build*.swz	Signed and unsigned RSLs containing Spark skins classes for MX components.
textLayout_*version.build*.swf textLayout_*version.build*.swz	Signed and unsigned RSLs containing TLF-related classes.

The SWZ and SWF files are files that you deploy. The SWC files in the framework/libs directory are the files that you compile your application against.

The signed framework RSLs are optimized, as described in "Optimizing RSL SWF files" on page 273. This means that they do not include debugging information. Flash Builder automatically uses unoptimized debug RSLs when you launch the debugger. If you compile on the command line, however, and use the command line debugger, you must either disable RSLs or specify non-optimized RSLs to use. Otherwise, you will not be able to set break points or take advantage of other debugging functionality.

Configuring framework RSLs

The Flex compilers compile against the framework RSLs by default. This means that all classes in the Flex framework are dynamically linked into your application SWF file, which should result in a smaller SWF file than if you statically linked in the library. In most cases, you do not have to specify which RSLs to use, or what order to use them in. The Flex compiler handles this for you.

You can view the order of preference for the location of these RSLs and their failovers in the flex-config.xml file. The preference is to use hosted, signed RSLs first, with local signed RSLs second.

The following example shows an entry in the configuration file that loads one of the framework RSLs:

```
<runtime-shared-library-path>
    <path-element>libs/framework.swc</path-element>
    <rsl-url>${hosted.rsl.url}/flex/${build.number}/framework_${build.number}.swz</rsl-url>
    <policy-file-url>${hosted.rsl.url}/crossdomain.xml</policy-file-url>
    <rsl-url>framework_${build.number}.swz</rsl-url>
    <policy-file-url></policy-file-url>
</runtime-shared-library-path>
```

The configuration file uses a `{hosted.rsl.url}` token in the first `<rsl-url>` element. The compiler replaces this with the location of the signed framework RSLs on the Adobe web site. If the client does not have network access, Flash Player then attempts to load the RSL that is specified by the next `<rsl-url>` element in the list. In this case, it is the SWZ file in the same directory as the application.

The configuration file also uses a `{build.number}` token in the name of the RSLs. The compiler replaces this with a build number during compilation. The name of the framework RSL depends on the build number of Flex that you are using.

You can also manually specify framework RSLs for your applications. You do this by compiling against the SWC files in your /frameworks/libs directory with the `runtime-shared-library-path` option on the command line. You can optionally add a policy file URL if necessary, plus one or more failover RSLs and their policy file URLs.

The following example compiles SimpleApp with the framework RSL:

```
mxmlc -runtime-shared-library-path=libs/framework.swc,
    framework_4.6.0.swz,,framework_4.6.0.swf
    SimpleApp.mxml
```

This example sets the signed framework RSL (*.swz) as the primary RSL, and then the unsigned framework RSL (*.swf) as the secondary RSL. This example does not specify a location for the policy file, so it is assumed that either the RSLs and the application are in the same domain or the policy file is at the root of the target server.

You can also specify the unsigned framework RSL as the primary RSL. This is common if you want to include debug information in the RSL, such as when you are using the command line to compile your SWF file and the command line debugger to debug it. Flash Builder automatically uses debug RSLs when you debug with it so you do not have to change any compiler options.

You can specify a signed SWZ file as the framework RSL and not specify an unsigned SWF file as a failover RSL. In this case, the application will not work in any Flash Player of version earlier than 9.0.115. In the HTML wrapper, you can detect and upgrade users to the newest Player by using the Express Install feature. For more information, see “Using Express Install in the wrapper” on page 2558. (Flex 4 applications require Player 10.0 or later, so this might not be an issue for you.)

If you use locally-hosted signed framework RSLs when you deploy your application, you must deploy the SWZ file to the location that you specified on the command line. You must also be sure that the crossdomain.xml file is in place at the RSL’s domain. To ensure that your application can support older versions of Flash Player, you should also deploy the unsigned framework RSL SWF file (in addition to the signed SWZ file), and specify that file as a failover RSL.

Disabling framework RSLs

The use of framework RSLs is enabled by default. You can disable RSLs when you compile your application by setting the `static-link-runtime-shared-libraries` compiler option to `true`, as the following example shows:

```
mxmlc -static-link-runtime-shared-libraries=true MyApp.mxml
```

Framework RSL digests

After the framework RSLs are transferred across the network, Flash Player generates a digest of the framework RSL and compares that digest to the digest that was stored in the application when it was compiled. If the digests match, then the RSL is loaded. If not, then Flash Player throws an error and attempts to load a failover RSL.

About the Player cache

The Player cache stores signed RSLs, such as the framework RSLs. You can manage the settings of the Player cache with the Settings Manager. The use of the RSLs in the Player cache is secure; no third party can inject code that will be executed. Only Adobe can sign RSLs; therefore, only Adobe RSLs can be stored in the Player cache.

The default size of the framework cache is 20MB. When the aggregate size of the cached RSLs in this directory meets or exceeds 20MB, Flash Player purges the cache. Files are purged on a least-recently-used basis. Less-used files are purged before files that have been used more recently. Purging continues until the cache size is below 60% of the maximum size. By default, this is 12MB.

The Global Storage Settings panel in the Settings Manager lets the user turn off the caching feature and increase or decrease its size. The Settings Manager is a special control panel that runs on your local computer but is displayed within and accessed from the Adobe website. If the user disables the Player cache, then Flash Player will not load SWZ files. Flash Player will load failover RSLs instead.

The following table shows the locations of the Player cache on various platforms:

Platform	Location
Windows 95/98/ME/2000/XP	C:\Documents and Settings*user_name*\Application Data\Adobe\Flash Player\AssetCache\
Windows Vista	C:\Users\user_name\AppData\Roaming\Adobe\Flash Player\AssetCache\
Linux	/home/*user_name*/.adobe/Flash_Player/AssetCache/
Mac OSX	/Users/*user_name*/Library/Cache/Adobe/Flash Player/AssetCache/

Using the framework RSLs in Flash Builder

Framework RSLs are enabled by default in Flash Builder. You can use the information in this section if you want to specify that only some of them are used, reorder the failover options, or customize some other aspect of the framework RSLs.

You can use Flash Builder to organize your RSLs and decide which SWC files to use as RSLs and which to use as statically-linked libraries. You can also use Flash Builder to configure the deployment locations, digests, and other properties of RSLs. The signed and unsigned framework RSLs have already been created for you and optimized. By default, the Flex compiler dynamically links against these RSLs. For locally-hosted RSLs, you must also deploy them to the location that you specified at compile time.

You can set each SWC file that your project uses to use one of the following linkage types:

- Merged Into Code —Indicates that classes and their dependencies from this library are added to the SWF file at compile time. They are not loaded at run time. The result is a larger SWF file, but one with no external dependencies.
- RSL — Indicates that this library will be used as an RSL. When you compile your application, the classes and their dependencies in this library are externalized, but you compile against them. You must then make them available as an RSL at run time.
- External — Indicates that this library will be externalized from the application, but it will not be used as an RSL. This excludes all the definitions in a library from an application but provides compile time link checking. Typically, only playerglobal.swc has this setting in an application project, but you might also use it to externalize assets that are used as modules or dynamically-loaded SWF files.

By default, all SWC files in the library path use the same linkage as the framework.

Use the framework SWC files as RSLs in Flash Builder

1. Open your Flex project in Flash Builder.
2. Select the project in the Navigator and select Project > Properties.
3. Select the Flex Build Path option in the resource list at the left. The Flex Build Path dialog box appears.
4. Select the Library Path tab.
5. Select Runtime Shared Library (RSL) from the Framework Linkage drop-down list. This instructs the application compiler to use the framework SWC files as RSLs.
6. Click OK to save your changes.

In the release output directory, Flash Builder includes the framework's SWF and SWZ files for you to deploy if your users require locally-hosted RSLs.

You can view the settings of the SWC files that are included with Flex but whose classes are not included in the framework RSLs by expanding the list of libraries used by Flex in the Library Path tab. This list includes the automation.swc and qtp.swc files. You might want to compile your application against these SWC files as RSLs if your application will benefit from doing so. These SWC files are not optimized, so if you use them as RSLs, you should optimize them as described in "Optimizing RSL SWF files" on page 273.

Use other Adobe SWC files as RSLs in Flash Builder

1 On the Library Path tab, click the + next to Flex 4.x to expand the list of SWC files.

2 Select the SWC file that you want to use as an RSL and click the + to show its properties. For example, click the + next to the automation.swc file.

3 Click the Link Type property and click the Edit button. The default link type is "Merged into code", which means that the library is statically linked to your application by default.

4 If you are already using the framework.swc file as an RSL, you can select the Use Same Linkage As Framework option. This option lets you toggle the RSL's status on and off by changing only the framework.swc file's linkage status. Otherwise, select Runtime Shared Library (RSL) from the Link Type drop-down list.

   The RSL Options field is enabled.

5 Select the Verification type and the deployment locations of the RSL as described in "Compiling applications with standard and cross-domain RSLs" on page 270.

6 Click OK to save your changes.

In some cases, you might not want to deploy the framework RSLs in the same directory as your main application. You might have a central server where these RSLs are stored, and need to share them across multiple applications.

Customize the deployment location of the framework RSLs

1 Edit the framework.swc file's Link Type settings in the Library Path Item Options dialog box.

2 Select the Digests radio button, if it is not already selected.

   Notice the two default entries in the Deployment paths field: an SWZ file and an SWF file. The SWZ file is the signed RSL that can be stored in the framework cache. The SWF file is an unsigned SWF file that is a failover if the SWZ file cannot be loaded.

3 To change the deployment location of the framework RSLs, click the SWZ file and then click the Edit button. The Edit RSL Deployment Path dialog box appears.

   In the Deployment Path/URL field, enter the location of the SWZ file. For example, "http://www.remoteserver.com/rsls/framework_4.0.0.5902.swc". The application tries to load the RSL from this location at run time.

   The location can be relative to the application's SWF file, or a full URL to a remote server. If the RSL is on a remote server, you might also be required to point to the policy file.

   In the Policy file URL field, enter the location of the crossdomain.xml file. You do not need to enter anything in this field if the application's SWF file and the RSLs are served from the same domain, or if the crossdomain.xml file is at the target server's root.

4 Click OK to save your changes in the Edit RSL Deployment Path dialog box.

5 Repeat the previous two steps for the SWF file, if you are also deploying the unsigned framework RSL with your application. If you do not deploy the failover SWF file, then users with Flash Player versions earlier than 9.0.115 will not be able to use your application. (Flex 4 applications require Player 10.0 or later, so this might not be an issue for you.)

6 Click OK to save your changes to the Library Path Item Options dialog box.

7 Ensure that the Verify RSL Digests option on the Library Path tab is selected.

8 Click OK to save your changes to the Flex Build Path settings.

When you deploy your application, you must manually copy the framework RSLs to the path that you specified on the remote server.

Example of using the framework RSLs on the command line

Framework RSLs are enabled by default for the command-line compiler. You can use the information in this section if you want to specify that only some of them are used, reorder the failover options, or customize some other aspect of the framework RSLs.

Create an application with a single Button control. After you create the application, determine the size of the application's compiled SWF file before you use the framework RSL. This will give you an idea of how much memory you are saving by using the framework RSL.

Compile the application as you normally would, with one exception. Add the `static-link-runtime-shared-libraries=true` option; this ensures that you are not using the framework RSL when compiling the application, regardless of the settings in your configuration files. Instead, you are compiling the framework classes into your application's SWF file.

With the mxmlc command-line compiler, use the following command to compile the application:

```
mxmlc -static-link-runtime-shared-libraries=true bin/SimpleApp.mxml
```

Examine the output size of the SWF file. Even though the application contains only a Button control, the size of the SWF file should be over 100KB. That is because it not only includes component and application framework classes, but it also includes all classes that those classes inherit from plus other framework dependencies. For visual controls such as a Button, the list of dependencies can be lengthy. If you added several more controls to the application, you will notice that the application does not get much larger. That is because there is a great deal of overlap among all the visual controls.

Run the application in a browser either from the file system or from a server. You can also run the application in the standalone player.

Next, compile the application again, but this time add the signed framework RSL as an RSL and the unsigned framework RSL as a failover RSL. For example:

```
mxmlc -runtime-shared-library-path=c:/p4/flex/flex/sdk/frameworks/libs/framework.swc,
    framework_4.0.0.5902.swz,,framework_4.0.0.5902.swf rsls/SimpleApp.mxml
```

The result is a SWF file that should be smaller than the previous SWF file.

This command includes a blank entry for the policy file URL. In this example, the crossdomain.xml file is not needed because you will deploy the RSLs and the application to the same domain, into the same directory.

In addition, the `runtime-shared-library-path` option includes an unsigned failover RSL as its final parameter. This is a standard framework RSL SWF file that is provided to support older versions of Flash Player that do not support signed RSLs. It is not cross domain, and if used by the client, it is stored in the browser's cache, not the framework cache.

Deploy the application and the framework RSLs to a server. You cannot request the SimpleApp.swf file from the file system because it loads network resources (the framework RSLs). This causes a security sandbox violation unless the application is loaded from a server. Deploy the SimpleApp.swf, framework_4.0.0.5902.swz, and framework_4.0.0.5902.swf files to the same directory on your web server.

Note: *You can point to the SWZ files that are hosted on the Adobe web site, rather than deploy your own SWZ files as RSLs. In this case, view the default entries for the RSLs in the flex-config.xml file to see how to link to them.*

Request the application in the browser or create a wrapper for the application and request that file.

To verify that the signed framework RSL was loaded by your application, you can look for an SWZ file in your framework cache. For more information, see "About the Player cache" on page 263.

After the client downloads the signed framework RSL, they will not have to download that RSL again for any application that uses signed framework RSLs, unless a new version of the framework is released or the RSL's SWZ file is purged from the Player cache.

Using standard and cross-domain RSLs

Standard and cross-domain RSLs are RSLs that you create from your custom component libraries. These RSLs are different from signed framework RSLs in that they are unsigned and can only be stored in the browser's cache. They are never stored in the Player cache.

To use standard or cross-domain RSLs, you perform the following tasks:

Create a library An RSL is created from a library of custom classes and other assets. You can create a library with either the Flash Builder Library Project or the compc command-line compiler. You can output the library as a SWC file or an open directory. The library includes a library.swf file and a catalog.xml file; the library.swf file is deployed as the RSL. For more information, see "Creating libraries" on page 276.

Compile your application against the library When you compile your application, you externalize assets from your application that are defined in the RSL. They can then be linked at run time rather than at compile time. You do this when you compile the application by passing the compile-time location of the library SWC file as well as the run-time location of the library's SWF file. For more information, see "Compiling applications with standard and cross-domain RSLs" on page 270.

Optimize the RSL After you generate a library and compile your application against it, you should run the optimizer against the library's SWF file. The optimizer reduces the SWF file by removing debugging code and unneeded metadata from it. While this step is optional, it is best practice to optimize a library SWF file before deploying it. For more information, see "Optimizing RSL SWF files" on page 273.

Deploy the RSL After you have compiled and optionally optimized your RSL, you deploy the library.swf file with your application so that it is accessible at run time. If the RSL is a cross-domain RSL, then you might also be required to deploy a crossdomain.xml file.

About standard RSLs

Standard RSLs can only be used by applications that are in the same domain as the RSL. You can benefit from using standard RSLs if you meet all of the following conditions:

- You host multiple applications in the same domain.
- You have custom component libraries.
- More than one application uses those custom component libraries.

Not all applications can benefit from standard RSLs. Applications that are in different domains or that do not use component libraries will not benefit from standard RSLs.

Standard RSLs can benefit from digests. While they do not require digests, you can use them to ensure that your application loads the latest RSL. For more information, see “About RSL digests” on page 269.

The following is a list of typical applications that can benefit from standard RSLs:

- Large applications that load multiple smaller applications that use a common component library. The top-level application and all the subordinate applications can share components that are stored in a common RSL.
- A family of applications on a server built with a common component library. When the user accesses the first application, they download an application SWF file and the RSL. When they access the second application, they download only the application SWF file (the client has already downloaded the RSL, and the components in the RSL are used by the two applications).
- A single monolithic application that changes frequently, but has a large set of components that rarely change. In this case, the components are downloaded once, while the application itself might be downloaded many times. This might be the case with charting components, where you might have an application that uses you change frequently, but the charting components themselves remain fairly static.

About cross-domain RSLs

Cross-domain RSLs can be used by applications in any domain or sub-domain. The benefits of cross-domain RSLs are the same as standard RSLs, but they are not restricted to being in the same domain as the application that loads them. This lets you use the same RSL in multiple applications that are in different domains.

To use a cross-domain RSL that is located on a remote server, the remote server must have a crossdomain.xml file that allows access from the application’s domain. The easiest way to do this is to add a crossdomain.xml file to the server’s root. To ensure that applications from any domain can access the RSL SWF file, you can use an open crossdomain.xml file such as the following:

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all"/>
    <allow-access-from domain="*" to-ports="*"/>
</cross-domain-policy>
```

This is not a best practice, however, because it allows requests from any domain to load the RSL, and other assets, from your server. You should instead restrict requests to only those domains that you trust by narrowing the entries in the `domain` attribute. You should also consider using a more restrictive meta-policy with the `permitted-cross-domain-policies` attribute. For more information, see “Using cross-domain policy files” on page 125.

You can store the crossdomain.xml file anywhere on the target server. When you compile a cross-domain RSL, you can specify the location of the crossdomain.xml file, and the application will look to that location to get permission to load the RSL. If you do not specify the location of the crossdomain.xml file when you compile your application, the application looks in the server’s root directory by default.

Cross-domain RSLs can fail to load into an application under the following conditions:

- The server on which the RSL is located fails
- The network fails, so remote files cannot be loaded
- The digest of the RSL when the application was compiled does not match the digest of the RSL when it is loaded
- The policy file is absent from the RSL’s server, or that server has a restrictive meta-policy

Cross-domain RSLs support a backup mechanism where a failover RSL can be loaded in the case of a server failure. If the server on which the main RSL fails, Flash Player will try to load a failover RSL whose location you specify when you compile the application.

About RSL digests

To ensure that the cross-domain RSL is coming from the trusted party, Flash Player reads the bytes of the incoming RSL and computes a one-way hash, or *digest*. The digest must match the digest that was stored in the application at compile time when the application was linked to the cross-domain RSL. If the RSL's digest matches the known digest, then Flash Player loads the cross-domain RSL. If the digests do not match, Flash Player displays an error message and does not load the RSL.

You can also use digests for standard RSLs. This is useful if you update your RSLs frequently, and want to ensure that the application loads the latest RSL.

To create a digest while compiling a library, you set the `compute-digest` compiler option to `true`. You can set this value in the flex-config.xml file, as the following example shows:

```
<compute-digest>true</compute-digest>
```

The default value of the `compute-digest` option is `true`.

In Flash Builder, to disable digests, you must add the following to the Additional Compiler Arguments field in the Flex Library Compiler dialog box:

```
-compute-digest=false
```

The compiler writes the digest inside the `swc/libraries/library/digests` element of the RSL's catalog.xml file. The following example shows the structure of `digest` elements, which are optional children of the `digests` element in the catalog.xml file:

```
<digests>
    <digest type="SHA-256" signed="true"
        value="d604d909d8d6d358097cf2f4ebf4707faf330469ed6b41dcdc5aaf6f4dd3bea9"/>
    <digest type="SHA-256" signed="false"
        value="d604d909d8d6d358097cf2f4ebf4707faf330469ed6b41dcdc5aaf6f4dd3bea9"/>
</digests>
```

The following table describes the tag and its attributes:

Option	Description
`digest`	Optional child of `digests` element.
`signed`	Whether the library is signed or not. This value is `true` if the digest is of a signed library and `false` otherwise. Only Adobe can create signed libraries. This attribute is required.
`type`	The kind of hash used to create the digest. The only currently-supported value is "SHA-256". This attribute is required.
`value`	The hash of the specified type. This is the digest of the RSL associated with the catalog.xml. This attribute is required.

When you compile your production application with an RSL that uses a digest for verification, set the `verify-digests` compiler option to `true` to indicate that the application must check the digest before using the RSL. The default value of this option is `true`.

If you have multiple RSLs, such as a main RSL plus a failover RSL, the compiler stores multiple digests inside the application.

The Flex compiler uses the SHA-256 digest algorithm from the java.security libraries to generate the digest.

About failover

The failover feature is used for two reasons:

- If the Flash Player is not at least version 9.0.115 and it tries to load a signed RSL, it will attempt to load the failover RSL. (Flex 4 applications require Player 10.0 or later, so this might not be an issue for you.)
- If a network or server failure occurs while loading the main RSL, Flash Player will attempt to load the failover RSL.

For framework RSLs, you typically specify a signed RSL as the main RSL, and an unsigned RSL as the failover RSL. When loading an application that uses signed framework RSLs, older versions of Flash Player skip the signed RSL and attempt to load the failover RSL, which is typically an unsigned RSL.

For all RSLs, the failover RSL provides a mechanism to load an RSL if the primary server is unavailable.

You can specify the location of the RSL and a failover RSL in your flex-config.xml file or on the command line as parameters to the `runtime-shared-library-path` option. The default failover RSL is framework_4.0.${build.number}.swf. In Flash Builder, you can add failover RSLs by adding them to the Deployment Paths in the Library Path Item Options dialog box.

Compiling applications with standard and cross-domain RSLs

Compiling your application with standard or cross-domain RSLs is easier to do in Flash Builder than it is on the command line. Flash Builder can automatically extract the RSL SWF file for you during the compilation process, and use that SWF to compile your application against. If you use the command-line compiler, then you must extract this file from the SWC file yourself prior to deployment. In either case, when you deploy the RSL SWF file, you should optimize it as described in “Optimizing RSL SWF files” on page 273.

Compile with standard or cross-domain RSLs in Flash Builder

1. Open your application’s project.
2. Select Project > Properties.
3. Select Flex Build Path in the option list. The Flex Build Path settings appear.
4. Select the Library Path tab.
5. Click the Add SWC button. The Add SWC dialog box appears.
6. Enter the path to the SWC file or click the Browse button to find the SWC file.
7. Click OK to add the SWC file to your project’s library path. Flash Builder adds the new SWC file to the list of SWC files in the Library Path tree.
8. Expand the SWC file in the Build Path Libraries tree by clicking the + next to it.
9. Change the Link Type to “Runtime shared library (RSL)”. You do this by selecting the Link Type and clicking the Edit button. When you change the Link Type to an RSL, Flash Builder displays additional options in the Library Path Item Options dialog box.
10. For Verification, select Digests if you are using a cross-domain RSL or if you are using a standard RSL that you want to verify at run time with a digest.

    Click the Add button to add the deployment path of the RSL SWF file. The Edit RSL Deployment Path dialog box appears.

    In the Deployment Path/URL field, enter the location that you will deploy the RSL’s SWF file to. This path is relative to the main application’s SWF file. The default is to deploy the RSL in the same directory as the main application.

    (Cross-domain RSLs only) Add a policy file URL if the application and the RSL will be on separate domains, and the crossdomain.xml file is not at the root of the RSL’s server.

You can have Flash Builder extract the RSL's SWF file from the SWC file automatically or you can do it manually. Deselect the Auto Extract swf to Deployment Path option if you want to extract the RSL's SWF file from the SWC file manually. Otherwise, Flash Builder will extract the SWF file for you. In general, you should manually extract the SWF file and optimize it before deploying it. Otherwise, the library.swf file will be larger than necessary. For more information on optimizing the RSL, see "Optimizing RSL SWF files" on page 273.

Click OK to save your changes.

11 Select None for Verification if you are using a standard RSL that does not use a digest. In this case, the RSL is assumed to not be cross-domain, and you therefore cannot specify a policy file URL. This is because standard RSLs must be deployed in the same domain as the main application, so policy files are not necessary. The deployment path of the RSL can also only be a local path that is relative to the location of the main application's SWF file.

12 Click OK to save your Library Path Item Options.

13 If the RSL is a cross-domain RSL, ensure that the Verify RSL Digests option is selected in the Library Path tab. If the RSL is standard and you do not want to use a digest, deselect this option. This option is overridden by the Verification setting in the Library Path Item Options dialog box.

14 Click OK to save your changes to the Flex Build Path settings.

Compile with standard or cross-domain RSLs on the command line

To use standard or cross-domain RSLs in your application on the command line, you use the `runtime-shared-library-path` application compiler option. This option has the following syntax:

```
-runtime-shared-library-path=path-element,rsl-url[,policy-file-url,failover-url,...]
```

The following table describes the `runtime-shared-library-path` arguments:

Argument	Description
`path-element`	Specifies the location of the SWC file or open directory to compile against. For example, c:\flexsdk\frameworks\libs\framework.swc. The compiler provides compile-time link checking by using the library specified by this option. This argument is required.
`rsl-url`	Specifies the location of the RSL SWF file that will be loaded by the application at run time. The compiler does not verify the existence of the SWF file at this location at compile time. It does store this string in the application, however, and uses it at run time. As a result, the SWF file must be available at run time but not necessarily at compile time. You specify the location of the SWF file relative to the deployment location of the application. For example, if you store the library.swf file in the *web_root*/libraries directory on the web server, and the application in the web root, you specify libraries/library.swf for the RSL SWF file. This argument is required. You must know the deployment location of the RSL SWF file relative to the application when you compile it. You do not have to know the deployment structure when you create the library SWC file, though, because components and classes are compiled into a SWC file and can be used by any application at compile time. The value of the `rsl-url` argument can be a relative URL, such as "../libraries/library.swf", or an absolute URL, such as "http://www.mydomain.com/libraries/library.swf". If it is on a different server, it must be a cross-domain or framework RSL. Standard RSLs can only be loaded from the same domain as the application. The default name of the RSL SWF file is library.swf, but you can change it to any name you want after you extract it from the SWC file. If you change it, then you must change the name you specify in the `rsl-url` option.
`policy-file-url`	Specifies the location of the policy file (crossdomain.xml) that gives permission to load the RSL from the server. For example, http://www.mydomain.com/rsls/crossdomain.xml. This is only necessary if the RSL and the application that uses it are on different domains. If you are serving the application and the RSL SWF file from the same domain, then you do not need to specify a policy file URL. This argument is optional. If you do not provide a value, then Flash Player looks at the root of the target web server for the crossdomain.xml file. For more information on using RSLs in different domains, see "About cross-domain RSLs" on page 268.
`failover-url`	Specifies the location of a secondary RSL if the first RSL cannot be loaded. This is most commonly used to ensure that an unsigned framework RSL will be used if the signed framework RSL fails to load. If the version of Flash Player is earlier than 9.0.115, it cannot load signed RSLs, so it must load an unsigned RSL. (Flex 4 applications require Player 10.0 or later, so this might not be an issue for you.) While this argument is used primarily to ensure that the framework RSL is loaded, it can also be used by cross-domain RSLs to ensure that a secondary RSL is available in case of network or server failure. This argument is optional. If you specify a second `policy-file-url`, then Flash Player will look to that location for the crossdomain.xml file for the failover RSL.

The following example shows how to use the `runtime-shared-library-path` option when compiling your application on the command line:

```
mxmlc -runtime-shared-library-path=../lib/mylib.swc,../bin/library.swf Main.mxml
```

Do not include spaces between the comma-separated arguments of the `runtime-shared-library-path` option on the command line.

Your application can use any number of RSLs. In Flash Builder, you add a list of RSLs on the Library Path tab. When using the command line, you add additional `runtime-shared-library-path` options. In both cases, the order of the RSLs is significant because base classes must be loaded before the classes that use them.

You can also use a configuration file to use RSLs, as the following example shows:

```
<runtime-shared-library-path>
    <path-element>../lib/mylib.swc</path-element>
    <rsl-url>../bin/library.swf</rsl-url>
</runtime-shared-library-path>
```

In the previous example, the file structure at compile time looks like this:

```
c:/Main.mxml
c:/lib/CustomDataGrid.swc
```

The deployed files are structured like this:

```
web_root/Main.swf
web_root/bin/library.swf
```

If you are using a cross-domain RSL, you can also specify the location of the crossdomain.xml file, and the location of one or more RSLs to be used as a failover RSL. The following example specifies a full URL for the location of the RSL, and the locations of a crossdomain.xml file and failover RSL on the command line:

```
mxmlc -runtime-shared-library-path=
    ../lib/mylib.swc,
    http://www.my-domain.com/rsls/library.swf,
    http://www.my-domain.com/rsls/crossdomain.xml,
    http://www.my-other-domain.com/rsls/library.swf,
    http://www.my-other-domain.com/rsls/crossdomain.xml
    Main.mxml
```

In the configuration file, this would be represented as follows:

```
<runtime-shared-library-path>
    <path-element>../lib/mylib.swc</path-element>
    <rsl-url>http://www.my-domain.com/rsls/library.swf</rsl-url>
    <policy-file-url>http://www.my-domain.com/rsls/crossdomain.xml</rsl-url>
    <rsl-url>http://www.my-other-domain.com/rsls/library.swf</rsl-url>
    <policy-file-url>http://www.my-other-domain.com/rsls/crossdomain.xml</rsl-url>
</runtime-shared-library-path>
```

Toggling RSLs on the command line

When compiling an application that uses RSLs, the command-line compiler options can be unwieldy and difficult to read. It is generally easier to define RSLs in your configuration files. However, when you do that, it is not very easy to enable or disable them as you develop your application because you have to edit the configuration file any time you want to change the way the RSLs are compiled.

To enable the use of RSLs without editing the configuration file, set the value of `static-link-runtime-shared-libraries` to `false`. By doing this, you can toggle the use of RSLs from the command line without having to edit the configuration file or enter long sets of command-line options. The default value of this option is `true`.

While you typically set the value of the `static-link-runtime-shared-libraries` option on the command line, you can also set it in the configuration file. If you set any RSL values on the command line, then the value of the `static-link-runtime-shared-libraries` option in the configuration file is ignored.

Optimizing RSL SWF files

The default SWF file in your SWC files includes debugging code and metadata that increase the file size. The debugging code is necessary for the compiler to run, but is not necessary at run time unless you want to debug the RSL. You can remove the debug code and unnecessary metadata by using the optimizer tool. This should result in a smaller RSL.

If you use the optimizer tool, you must keep track of two separate library files: one for compiling (the larger, pre-optimized one) and one for deploying (the smaller, optimized one). You compile your main application against the non-optimized library but then deploy the optimized library so that it can be loaded at run time.

The optimizer tool is in the bin directory. For Unix and Mac OS, it is a shell script called optimizer. For Windows, it is optimizer.exe. You invoke it only from the command line. The Java settings are managed by the jvm.config file in the bin directory.

The syntax for the optimizer tool is as follows:

```
optimizer -keep-as3-metadata Bindable Managed ChangeEvent NonCommittingChangeEvent Transient
-input input_swf -output output_swf
```

You must specify the `keep-as3-metadata` option and pass it the required metadata. At a minimum, you should specify the Bindable, Managed, ChangeEvent, NonCommittingChangeEvent, and Transient metadata names. You can also specify custom metadata that you want to remain in the optimized SWF file.

You can specify the configuration file that the optimizer tool uses by using the `load-config` option.

To get help while using the optimizer tool, enter the `-help list advanced` option:

```
optimizer -help list advanced
```

By default, the optimizer tool saves an optimized version of the SWF file in the current directory, with the name output.swf. This file should be smaller than the library.swf because it does not contain the debugging code.

Use optimized RSLs

1 Create an RSL by compiling a library project in Flash Builder or building a SWC file with the compc command line tool.

2 Compile your main application and reference the RSL.

3 Extract the library.swf file from your RSL’s SWC file, if you haven’t done so already.

4 Run the optimizer against the library.swf file, for example:

```
optimizer -keep-as3-metadata Bindable Managed ChangeEvent NonCommittingChangeEvent
Transient -input c:\rsls\library.swf -output c:\rsls\output\output.swf
```

5 Run the digest tool against the optimized library.swf file; for example:

```
digest -digest.rsl-file c:\rsls\output\output.swf -digest.swc-path c:\rsls\output\
```

6 Deploy the optimized library (in this case, output.swf) with the application so that the application uses it at run time.

Example of creating and using a standard and cross-domain RSL

This example walks you through the process of creating a library and then using that library as a standard and a cross-domain RSL with an application. It uses the command-line compilers, but you can apply the same process to create and use RSLs with a Flash Builder project. This example first shows you how to use a standard RSL, and then how to use that same RSL as a cross-domain RSL.

Keep in mind that a SWC file is a library that contains a SWF file that contains run-time definitions and additional metadata that is used by the compiler for dependency tracking, among other things. You can open SWC files with any archive tool, such as WinZip, and examine the contents.

Before you use an RSL, first learn how to statically link a SWC file. To do this, you build a SWC file and then set up your application to use that SWC file.

In this example you have an application named app.mxml that uses the ProductConfigurator.as and ProductView.as classes. The structure of this example includes the following files and directories:

```
project/src/app.mxml
project/libsrc/ProductConfigurator.as
project/libsrc/ProductView.as
project/lib/
project/bin/
```

To compile this application without using libraries, you can link the classes in the /libsrc directory using the `source-path` option, as the following example shows:

```
cd project/src
mxmlc -o=../bin/app.swf -source-path+=../libsrc app.mxml
```

This command adds the ProductConfigurator and ProductView classes to the SWF file.

To use a library rather than standalone classes, you first create a library from the classes. You use the compc compiler to create a SWC file that contains the ProductConfigurator and ProductView classes, as the following command shows:

```
cd project
compc -source-path+=libsrc -debug=false -o=lib/mylib.swc ProductConfigurator ProductView
```

This compiles the mylib.swc file in the lib directory. This SWC file contains the implementations of the ProductConfigurator and ProductView classes.

To recompile your application with the new library, you add the library with the `library-path` option, as the following example shows:

```
cd project/src
mxmlc -o=../bin/app.swf -library-path+=../lib/mylib.swc app.mxml
```

This links the library at compile time, which does not result in any benefits of externalization. The library is not yet being used as an RSL. If you use the new library as an RSL, the resulting SWF file will be smaller, and the library can be shared across multiple applications.

Now you can recompile the application to use the library as an external RSL rather than as a library linked at compile time.

The first step is to compile your application with the `runtime-shared-library-path` option. This option instructs the compiler to specifically exclude the classes in your library from being compiled into your application, and provides a run time location of the RSL SWF file.

```
cd project/src
mxmlc -o=../bin/app.swf -runtime-shared-library-path=../lib/mylib.swc,myrsl.swf app.mxml
```

The next step is to prepare the RSL so that it can be found at run time. To do this, you extract the library.swf file from the SWC file with any archive tool, such as WinZip or jar.

The following example extracts the SWF file by using the unzip utility on the command line:

```
cd project/lib
unzip mylib.swc library.swf
mv library.swf ../bin/myrsl.swf
```

This example renames the library.swf file to myrsl.swf and moves it to the same directory as the application SWF file.

The next step is to optimize the library.swf file so that it does not contain any debug code or unnecessary metadata. The following example optimizes the SWF file by using the optimizer tool:

```
optimimzer -keep-as3-metadata Bindable Managed ChangeEvent NonCommittingChangeEvent Transient
    -input bin/myrsl.swf -output bin/myrsl.swf
```

You now deploy the main application and the RSL. In this example, they must be in the same directory. If you want to deploy the myrsl.swf file to a different directory, you specify a different path in the `runtime-shared-library-path` option.

You can optionally run the optimizer tool on the myrsl.swf file to make it smaller before deploying it. For more information, see "Optimizing RSL SWF files" on page 273.

To use the RSL as a cross-domain RSL, you add a crossdomain.xml file, a failover RSL, and its crossdomain.xml file to the option, as the following example shows:

```
cd project/src
    mxmlc -o=../bin/app.swf -runtime-shared-library-path=../lib/mylib.swc,
    http://www.my-remote-domain.com/rsls/myrsl.swf,
    http://www.my-remote-domain.com/rsls/crossdomain.xml,
    http://www.my-other-remote-domain.com/rsls/myrsl.swf,
    http://www.my-other-remote-domain.com/rsls/crossdomain.xml,
    Main.mxml
```

Next, you create a crossdomain.xml file. If the domain that are you running the application on is my-local-domain.com, then you can create a crossdomain.xml file that looks like the following:

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all"/>
    <allow-access-from domain="*.my-local-domain.com" to-ports="*"/>
</cross-domain-policy>
```

You now deploy the main application and the RSL. This time, however, the RSL is deployed on a remote server in the /rsls directory. You must also ensure that the crossdomain.xml file is in that directory. Finally, you must ensure that the failover RSL and its crossdomain.xm l file are deployed to the other remote domain.

Creating libraries

To use standard or cross-domain RSLs, you must first create the library that will be used as an RSL. If you want to use framework RSLs, the libraries are already created for you. In that case, all you need to do is compile against them and then deploy the SWZ or SWF file with your application.

A standard or cross-domain RSL is a library of custom components, classes, and other assets that you create. You can create a library using either Flash Builder or the compc command-line compiler. A library is a SWC file or open directory that contains a library.swf file and a catalog.xml file, as well as properties files and other embedded assets. You can use any library as an RSL, but libraries do not need to be used as RSLs.

Creating libraries in Flash Builder

In Flash Builder, you create a new library by selecting File > New > Flex Library Project. You add resources to a library by using the Flex Library Build Path dialog box. When you select Build Library or Build All, Flash Builder creates a SWC file. This SWC file contains the library.swf file.

Creating libraries on the command line

On the command line, you create a library by using the compc compiler. You add files to a library by using the `include-classes` and `include-namespaces` options when you compile the SWC file.

The following example creates a library called CustomCellRenderer.swc with the compc compiler:

```
compc -source-path ../mycomponents/components/local
    -include-classes CustomCellRendererComponent
    -directory=true
    -debug=false
    -output ../libraries/CustomCellRenderer
```

The options on the command line can also be represented by settings in the flex-config.xml file, as the following example shows:

```
<?xml version="1.0">
<flex-config>
    <compiler>
        <source-path>
            <path-element>../mycomponents/components/local</path-element>
        </source-path>
    </compiler>
    <output>../libraries/CustomCellRenderer</output>
    <directory>true</directory>
    <debug>false</false>
    <include-classes>
        <class>CustomCellRendererComponent</class>
    </include-classes>
</flex-config>
```

All classes and components must be statically linked into the resulting library. When you use the compc compiler to create the library, do not use the `include-file` option to add files to the library, because this option does not statically link files into the library.

Optimizing libraries

Optimizing libraries means to remove debugging and other code from the library prior to deployment. For normal libraries that you are not using as RSLs, you do not need to optimize. This is because you will likely want to debug against the library during development, so you will need the debug code inside the library. And, when you compile the release version of your application, the compiler will exclude debug information as it links the classes from the library.

When you compile a library for production use as an RSL, however, you can set the `debug` compiler option to `false`. The default value is `true` for compc, which means that the compiler, by default, includes extra information in the SWC file to make it debuggable. You should avoid creating a debuggable library that you intend to use in production so that the RSL's files are as small as possible. If you set the value of the `debug` option to `false`, however, you will not be able to debug against the RSL for testing.

If you do include debugging information, you can still optimize the RSL after compiling it, which removes the debugging information as well as unnecessary metadata. For more information, see "Optimizing RSL SWF files" on page 273.

Before you deploy an RSL, you extract the SWF file that is inside the library SWC file and optimize it. The default name of this SWF file is library.swf. After you extract it from the SWC file, you can rename it to anything you want. When you deploy the RSL, you deploy the SWF file so that the application can load it at run time.

On the command line, you typically specify that the output of compiling a library be an open directory rather than a SWC file by using the `directory` option and the `output` option. The output is an open directory that contains the following files:

- catalog.xml
- library.swf

In addition, the library contains properties files and any images or other embedded assets that are used by the library.

If you do not specify that the output be an open directory, you must manually extract the library.swf file from the SWC file with a compression utility, such as PKZip.

In Flash Builder, you can instruct the compiler to automatically extract the SWF file for you when you add the RSL SWC file to the project. Doing this does not optimize the SWF file, so the library will be bigger than if you extract it yourself and optimize it.

To manually extract an RSL SWF file from its SWC file in Flash Builder, add the `output` and `directory` options to the Additional Compiler Arguments field in the Flex Compiler dialog box. In both cases, you specify the deployment location of the SWF file when you add the SWC file to your project as an RSL.

When creating a library, you need to know if you will be using the library as a standard RSL or as a cross-domain RSL. If you are using it as a cross-domain RSL, you must include the digest information in the library. For more information, see “About cross-domain RSLs” on page 268.

After you create the library, you then compile your application against the SWC file and specify the library’s SWF file’s location for use at run time. For more information, see “Compiling applications with standard and cross-domain RSLs” on page 270.

For more information on using the compc compiler options, see “Flex compilers” on page 2164.

Troubleshooting RSLs

RSLs can be complicated to create, use, and deploy. The following table describes common errors and techniques to resolve them:

Error	**Resolution**
`#1001 Digest mismatch with RSL`	This error indicates that the digest of a library does not match the RSL SWF file. When you compile an application that uses an RSL, you specify the library SWC file that the application uses for link checking at compile time and an RSL SWF file that the application loads at run time. The digest in the library’s catalog.xml file must match the digest of the RSL SWF file or you will get this error. If this error persists, recompile the application against the library SWC file again and redeploy the application’s SWF file. If you are using framework RSLs, then the SWZ file is a different version than what the application was compiled against. Check whether this is the case by adding a failover RSL SWF file and recompiling. If the error does not recur, then the SWZ file is out of sync. If you are creating an optimized RSL, be sure to run the digest tool against the library.swf file after you optimized the library.swf file.
`#2032` Stream Error	This error indicates that the SWZ or SWF file is not being found. The frequently occurs if the client is not able to download the signed framework RSL from the Adobe web site, and you have not deployed a local signed framework RSL. If the client has limited or not internet connectivity, be sure to deploy the local framework RSL to a location that is accessible to the client. For more information, see “Using the framework RSLs” on page 260. It can also occur if Flash Player tries to load a custom RSL that is not available. For example, if you specified only “mylib.swf” as the value of the `rsl-url` parameter to the `runtime-shared-library-path` option, but the SWF file is actually in a sub-directory such as /rsls, then you must either recompile your application and update the value of the `rsl-url` parameter to “rsls/mylib.swf”, or move mylib.swf to the same directory as the application’s SWF file.
`#2046`	This error indicates that the loaded RSL was not signed properly. In the case of framework RSLs, the framework’s SWZ file that the application attempted to load at run time was not a properly signed SWZ file. You must ensure that you deploy an Adobe-signed RSL, or be sure to use the signed framework RSLs that are deployed on the Adobe web site.

Error	**Resolution**

`#2048`	The cause of this error is that you do not have a crossdomain.xml file on the server that is returning the RSL. You should add a file to that server. For more information, see “Using cross-domain policy files” on page 125.
	If you are loading signed framework RSLs from the Adobe web site, you should not get this error.
	If you put the crossdomain.xml file at the server’s root, you do not have to recompile your application. This is because the application will look for that file at the server’s root by default, so there is no need to explicitly define its location.
	If you cannot store a crossdomain.xml file at the remote server’s root, but can put it in another location on that server, you must specify the file’s location when you compile the application. On the command line, you do this by setting the value of the `policy-file-url` argument of the `runtime-shared-library-path` option.
	In the following example, the RSL SWF file is located in the /rsls directory on www.domain.com. The crossdomain.xml file is located in the same directory, which is not the server’s root, so it must therefore be explicitly specified:
	`mxmlc -runtime-shared-library-path=   ../lib/mylib.swc,`
	`http://www.mydomain.com/rsls/myrsl.swf,`
	`http://www.mydomain.com/rsls/crossdomain.xml   Main.mxml`

`#2148`	This error occurs when you try to open an application that uses RSLs in the standalone player or in the browser by using the file system and not a server. It means that you are violating the security sandbox of Flash Player by trying to load file resources. You must deploy your application and RSLs to a network location, and request the application with a network request so that Flash Player will load the RSL. If you are testing the application locally, you can add the directory to your Player trust file to avoid this error.
`Requested resource not found`	You might find this error in your web server logs. If you deploy the RSL SWF file to a location other than that specified when you compiled the application, then you will get an error similar to this when the application tries to run. The solution is to either recompile your application and correct the deployment location of the RSL SWF file or to move the RSL SWF file to the location that the application expects.

Chapter 4: Building the user interface

Visual components

You use visual components to build Adobe® Flex® applications. Visual components have a flexible set of characteristics that let you control and configure them as necessary to meet your application's requirements.

About visual components

Flex includes a component-based development model that you use to develop your application and its user interface. You can use the prebuilt visual components included with Flex, extend components to add new properties and methods, and create components as required by your application.

Visual components are extremely flexible and provide you with a great deal of control over the component's appearance, how the component responds to user interactions, the font and size of any text included in the component, the size of the component in the application, and many other characteristics.

The characteristics of visual components include the following:

Size Height and width of a component. All visual components have a default size. You can use the default size, specify your own size, or let Flex resize a component as part of laying out your application.

Events Application or user actions that require a component response. Events include component creation, mouse actions such as moving the mouse over a component, and button clicks.

Styles Characteristics such as font, font size, and text alignment. These are the same styles that you define and use with Cascading Style Sheets (CSS).

Effects Visible or audible changes to the component triggered by an application or user action. Examples of effects are moving or resizing a component based on a mouse click.

Skins Classes that control a visual component's appearance.

Spark and MX components

Flex defines two sets of components: Spark and MX. The Spark components are new for Flex 4 and are defined in the spark.* packages. The MX components shipped in previous releases of Flex and are defined in the mx.* packages.

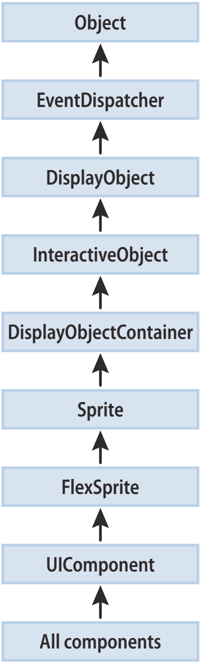
The main differences between Spark and MX components are how you use CSS styles with the components and how you skin them. For the container components, there are additional differences about how the containers perform layout.

Spark and MX define some of the same components. For example, Spark defines a button control in the spark.components package, and MX defines a button control in the mx.controls package. When a component is available in both Spark and MX, Adobe recommends that you use the Spark component.

Spark and MX also define components that are unique. For example, Spark defines components to perform three dimensional effects. MX defines data visualization components, such as the DataGrid and AdvancedDataGrid controls, not included in Spark. Your applications often contains a mixture of Spark and MX components.

Class hierarchy for visual components

Flex visual components are implemented as a class hierarchy in ActionScript. Therefore, each visual component in your application is an instance of an ActionScript class. The following image shows this hierarchy in detail up to the Object class:



All visual components are derived from the UIComponent class and its superclasses, the Flex Sprite through Object classes, and inherit the properties and methods of their superclasses. In addition, visual components inherit other characteristics of the superclasses, including event, style, and effect definitions.

Using the UIComponent class

The UIComponent class is the base class for all Flex visual components. For detailed documentation, see UIComponent in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Commonly used UIComponent properties

The following table lists only the most commonly used properties of components that extend the UIComponent class:

Property	**Type**	**Description**
`doubleClickEnabled`	Boolean	Setting to `true` lets the component dispatch a `doubleClickEvent` when a user presses and releases the mouse button twice in rapid succession over the component.
`enabled`	Boolean	Setting to `true` lets the component accept keyboard focus and mouse input. The default value is `true`. If you set `enabled` to `false` for a container, Flex dims the color of the container and all of its children, and blocks user input to the container and to all its children.
`height`	Number	The height of the component, in pixels. In MXML tags, but not in ActionScript, you can set this property as a percentage of available space by specifying a value such as 70%; in ActionScript, you must use the `percentHeight` property. The property always returns a number of pixels. In ActionScript, you use the `perCent`
`id`	String	Specifies the component identifier. This value identifies the specific instance of the object and should not contain any white space or special characters. Each component in a Flex document must have a unique `id` value. For example, if you have two custom components, each component can include one, and only one Button control with the id "okButton".
`percentHeight`	Number	The height of the component as a percentage of its parent container, or for the Application container, the full height of the browser. Returns `NaN` if a percent-based width has never been set or if a `height` property was set after the `percentHeight` was set.
`percentWidth`	Number	The width of the component as a percentage of its parent container, or for the Application container, the full span of the browser. Returns `NaN` if a percent-based width has never been set or if a `width` property was set after the `percentWidth` was set.
`styleName`	String	Specifies the style class selector to apply to the component.
`toolTip`	String	Specifies the text string displayed when the mouse pointer hovers over that component.
`visible`	Boolean	Specifies whether the container is visible or invisible. The default value is `true`, which specifies that the container is visible.
`width`	Number	The width of the component, in pixels. In MXML tags, but not in ActionScript, you can set this property as a percentage of available space by specifying a value such as 70%; in ActionScript, you must use the `percentWidth` property. The property always returns a number of pixels.
`x`	Number	The component's ***x*** position within its parent container. Setting this property directly has an effect only if the parent container uses absolute positioning.
`y`	Number	The component's ***y*** position within its parent container. Setting this property directly has an effect only if the parent container uses absolute positioning.

Using components in MXML and ActionScript

Every Flex component has an MXML API and an ActionScript API. A component's MXML tag attributes are equivalent to its ActionScript properties, styles, effects, and events. You can use both MXML and ActionScript when working with components.

Configure a component

1. Set the value of a component property, event, style, or effect declaratively in an MXML tag, or at run time in ActionScript code.
2. Call a component's methods at run time in ActionScript code. The methods of an ActionScript class are not exposed in the MXML API.

The following example creates a Spark Button control in MXML. When the user clicks the Button control, it updates the text of a Spark TextArea control by using an ActionScript function.

```
<?xml version="1.0"?>
<!-- components\ButtonApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function handleClick():void {
                text1.text="Thanks for the click!";
            }
        ]]>
    </fx:Script>
    <s:Button id="button1"
        label="Click here!"
        width="100"
        fontSize="12"
        click="handleClick();"/>
    <s:TextArea id="text1"/>
</s:Application>
```

This example has the following elements:

- The `id` property is inherited by the Button control from the UIComponent class. You use it to specify an identifier for the component. This property is optional, but you must specify it if you want to access the component in ActionScript.
- The `label` property is defined by the Button control. It specifies the text that appears in the button.
- The `width` property is inherited from the UIComponent class. It optionally specifies the width of the button, in pixels.
- The Button control dispatches a `click` event when a user presses and releases the main mouse button. The MXML `click` attribute specifies the ActionScript code to execute in response to the event.
- The `fontSize` style is inherited from the UIComponent class. It specifies the font size of the label text, in pixels.

Note: *The numeric values specified for font size in Flex are actual character sizes in pixels. These values are not equivalent to the relative font size specified in HTML by using the `<font>` tag.*

The `click` event attribute can also take ActionScript code directly as its value, without your having to specify it in a function. Therefore, you can rewrite this example as the following code shows:

```
<?xml version="1.0"?>
<!-- components\ButtonAppAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Button id="button1"
        label="Click here!"
        width="100"
        fontSize="12"
        click="text1.text='Thanks for the click!';"/>
    <s:TextArea id="text1"/>
</s:Application>
```

***Note:** Although you can specify multiple lines of ActionScript code (separated by semicolons) as the value of the `click` event attribute, for readability you should limit the `click` event to only one or two lines of code.*

Initializing components at run time

Flex uses MXML property attributes to initialize your components. However, you might want to use some logic to determine initial values at run time. For example, you might want to initialize a component with the current date or time. Flex must calculate this type of information when the application executes.

Every component dispatches several events during its life cycle. In particular, all components dispatch the following events that let you specify ActionScript to initialize a component:

preInitialize Dispatched when a component has been created in a rough state, and no children have been created.

initialize Dispatched when a component and all its children have been created, but before the component size has been determined.

creationComplete Dispatched when the component has been laid out and the component is visible (if appropriate).

You can use the `initialize` event to configure most component characteristics; in particular, use it to configure any value that affects the component's size. Use the `creationComplete` event if your initialization code must get information about the component layout.

The following example configures Flex to call the `initDate()` function when it initializes the Label control. When Flex finishes initializing the Label control, and before the application appears, Flex calls the `initDate()` function.

```
<?xml version="1.0"?>
<!-- components\LabelInit.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            private function initDate():void {
                label1.text += new Date();
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label id="label1"
            text="Today's Date: "
            initialize="initDate();"/>
    </s:VGroup>
</s:Application>
```

You can also express the previous example without a function call by adding the ActionScript code in the component's definition. The following example does the same thing, but without an explicit function call:

```
<?xml version="1.0"?>
<!-- components\LabelInitAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:VGroup>
        <s:Label id="label1"
            text="Today's Date:"
            initialize="label1.text += new Date();"/>
    </s:VGroup>
</s:Application>
```

As with other calls that are embedded within component definitions, you can add multiple ActionScript statements to the `initialize` MXML attribute by separating each function or method call with a semicolon. The following example calls the `initDate()` function and writes a message in the flexlog.txt file when the `label1` component is instantiated:

```
<?xml version="1.0"?>
<!-- components\LabelInitASAndEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            private function initDate():void {
                label1.text += new Date();
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label id="label1"
            text="Today's Date:"
            initialize="initDate(); trace('The label is initialized!');"/>
    </s:VGroup>
</s:Application>
```

By default, on Microsoft Windows, the flexlog.txt file is written to C:\Documents and Settings\USERNAME\Application Data\Macromedia\Flash Player\Logs. For the location of this file on other operating systems, see “Editing the mm.cfg file” on page 2224.

Configuring components: syntax summary

The following table summarizes the MXML and ActionScript component APIs that you use to configure components:

	MXML example	**ActionScript example**

Read-write property	`<s:Panel id="p1" title="My Title"`
	`visible="true"/>`
		`p1.title="My Title";`
		`tile1.visible=true;`

Read-only property	You cannot use a read-only property as an attribute in MXML.	To get the value of a read-only property: `var theClass:String=mp1.className;`
Method	Methods are not available in MXML.	`myList.sortItemsBy("data", "DESC");`

Event	`<mx:Accordion id="myAcc"`
	`change="changeHandler (event);"/>`
	You must also define a `changeHandler()` function as shown in the ActionScript example.
		`private function changeHandler(event:MouseEvent):void`
		`{...}`
		`myButton.addEventListener("click", changeHandler);`

Style	`<s:Panel id="p1"`
	`borderAlpha="0.5"`
	`borderColor="blue"/>`
		To set the style:
		`p1.setStyle("borderAlpha", 0.5);`
		`p1.setStyle("borderColor", 'blue');`
		To get the style:
		`var currentBorderAlpha:Number =`
		`p1.getStyle("borderAlpha");.`

Effect	`<s:Panel id="p1"`
	`show="scale.end();scale.play();"/>`
		Call the effect’s `end()` and `play()` methods in an event handler:.

Sizing visual components

The Flex sizing and layout mechanisms provide several ways for you to control the size of visual components:

Default sizing Automatically determines the sizes of controls and containers. To use default sizing, you do not specify the component’s dimensions or layout constraints.

Explicit sizing You set the `height` and `width` properties to pixel values. When you do this, you set the component dimension to absolute sizes, and Flex does not override these values. The following `<s:Application>` tag, for example, sets explicit Application dimensions:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="300"
    width="600">
```

Percentage-based sizing You specify the component size as a percentage of its container size. To do this, you specify the `percentHeight` and `percentWidth` properties, or, in an MXML tag, set the `height` and `width` properties to percentage values such as 100%. The following code, for example, sets percentage-based dimensions for an HGroup container:

```
<s:HGroup id="hGroup1"
    height="30%" width="90%"/>
```

The following ActionScript line resets the HGroup width to a different percentage value:

```
hGroup1.percentWidth=40;
```

Constraint-based layout You can control size *and* position by anchoring components sides, centers, or baselines to locations in their container by specifying the `top`, `bottom`, `left`, `right`, `baseline`, `horizontalCenter`, and `verticalCenter` styles. You can use constraint-based layout only for the children of a container that uses absolute layout. That includes all Spark containers and the MX Application, Panel, and Canvas containers. The following example uses constraint-based layout to position an HGroup horizontally, and explicit sizing and positioning to determine the vertical width and position:

```
<s:HGroup id="hGroup2"
    left="30"right="30"
    y="150"
    height="100"/>
```

You can mix sizing techniques; however, you must ensure that the mix is appropriate. Do not specify more than one type of sizing for a component dimension; for example, do not specify a `height` and a `percentHeight` for a component. Also, ensure that the resulting sizes are appropriate; for example, if you do not want scroll bars or clipped components, ensure that the sizes of a container's children do not exceed the container size.

For detailed information on how Flex sizes components, and how you can specify sizing, see "Laying out components" on page 359.

Examples of component sizing

The following example shows sizing within an explicitly sized container when some of the container's child controls are specified with explicit widths and some with percentage-based widths. It shows the flexibility and the complexities involved in determining component sizes. The application logs the component sizes to flashlog.txt, so you can confirm the sizing effect.

```
<?xml version="1.0"?>
<!-- components\CompSizing.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="logSizes();">
    <fx:Script>
        <![CDATA[
            private function logSizes():void {
                trace("HGroup: "+ hb1.width);
                trace("Label: "+ lb1.width);
                trace("Image: "+ img1.width);
                trace("Button: "+ b1.width);
            }
        ]]>
    </fx:Script>
    <s:HGroup id="hb1" width="250">
        <s:Label id="lb1"
            text="Hello"
            width="50"/>
        <mx:Image id="img1"
            source="@Embed(source='assets/flexlogo.jpg')"
            width="75%"/>
        <s:Button id="b1"
            label="Button"
            width="25%"/>
    </s:HGroup>
</s:Application>
```

The application consists of a 250-pixel-wide HGroup container that contains a 50-pixel-wide label, an image that requests 75% of the container width, and a button that requests 25% of the container width. The component sizes are determined as follows:

1 Flex reserves 50 pixels for the explicitly sized Label control.

2 Flex puts an 6-pixel gap between components in an HGroup container by default, so it reserves 12 pixels for the gaps; this leaves 188 pixels available for the two percentage-based components.

3 The minimum width of the Button component, if you do not specify an explicit width, fits the label text plus padding around it. In this case, the minimum size is 70 pixels. However, since you specified the width as 25%, Flex reserves 25% of the available 188 pixels, or 47 pixels, for the Button control.

4 The percentage-based image requests 75% of 188 pixels, or 141 pixels.

If you change the button and image `size` properties to 50%, each get 50% of the available space, or 94 pixels.

The following example uses explicit sizing for the Image control and default sizing for the Button control:

```
<?xml version="1.0"?>
<!-- components\CompSizingExplicit.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="logSizes();">
    <fx:Script>
        <![CDATA[
            private function logSizes():void {
                trace("HGroup: "+ hb1.width);
                trace("Label: "+ lb1.width);
                trace("Image: "+ img1.width);
                trace("Button: "+ b1.width);
            }
        ]]>
    </fx:Script>
    <s:HGroup id="hb1" width="250">
        <s:Label id="lb1"
            text="Hello"
            width="50"/>
        <mx:Image id="img1"
            source="@Embed(source='assets/flexlogo.jpg')"
            width="119" />
        <s:Button id="b1"
            label="Button"/>
    </s:HGroup>
</s:Application>
```

Percentage-based sizing removes the need to explicitly consider the gaps and margins of a container in sizing calculations, but its greatest benefit applies when containers resize. Then, the percentage-based children resize automatically based on the available container size. In the following example, the series of controls on the left resize as you resize your browser, but the corresponding controls on the right remain a fixed size because their container is fixed. Click the first Button control to log the component sizes to flashlog.txt.

```
<?xml version="1.0"?>
<!-- components\CompSizingPercent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            private function logSizes():void {
                trace("HGroup: "+ hb1.width);
                trace("Label: "+ lb1.width);
                trace("Image: "+ img1.width);
                trace("Button: "+ b1.width);
            }
        ]]>
    </fx:Script>
    <s:HGroup width="100%">
        <s:HGroup id="hb1" width="40%">
            <s:Label id="lb1"
                text="Hello"
                width="50"/>
            <mx:Image id="img1"
                source="@Embed(source='assets/flexlogo.jpg')"
                width="60%"/>
            <s:Button id="b1"
                label="Button"
                width="40%"
                click="logSizes();"/>
        </s:HGroup>

        <s:HGroup width="260">
            <s:Label
                text="Hello"
                width="50"/>
            <mx:Image
                source="@Embed(source='assets/flexlogo.jpg')"
                width="119" />
            <s:Button
                label="Button"/>
        </s:HGroup>
    </s:HGroup>
</s:Application>
```

For more information about sizing considerations, see “Sizing components” on page 365.

Handling events

Flex applications are event driven. *Events* let a programmer know when the user has interacted with an interface component, and also when important changes have happened in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing.

When an instance of a component dispatches an event, objects that have registered as *listeners* for that event are notified. You define event listeners, also called event *handlers*, in ActionScript to process events. You register event listeners for events either in the MXML declaration for the component or in ActionScript. For additional examples of the event handling, see “Using components in MXML and ActionScript” on page 282.

The following example registers an event listener in MXML that is processed when you change views in an Accordion container.

```
<?xml version="1.0"?>
<!-- components\CompIntroEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="300"
    height="280">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function handleAccChange():void {
                Alert.show("You just changed views.");
            }
        ]]>
    </fx:Script>
    <!-- The Accordion container dispatches a change event when the
            selected child container changes. -->
    <mx:Accordion id="myAcc"
        height="60"
        width="200"
        change="handleAccChange();">
        <s:NavigatorContent label="Box 1">
            <s:Label text="This is one view."/>
        </s:NavigatorContent>
        <s:NavigatorContent label="Box 2">
            <s:Label text="This is another view."/>
        </s:NavigatorContent>
    </mx:Accordion>
</s:Application>
```

You can pass an event object, which contains information about the event, from the component to the event listener.

For the Accordion container, the event object passed to the event listener for the `change` event is of class IndexChangedEvent. You can write your event listener to access the event object, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\CompIntroEventAcc.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="300"
    height="280">
    <fx:Script>
        <![CDATA[
            // Import the class that defines the event object.
            import mx.events.IndexChangedEvent;
            import mx.controls.Alert;

            private function handleChange(event:IndexChangedEvent):void {
                var currentIndex:int=event.newIndex;
                Alert.show("You just changed views.\nThe new index is "
                    + event.newIndex + ".");
            }
        ]]>
    </fx:Script>
    <!-- The Accordion control dispatches a change event when the
            selected child container changes. -->
    <mx:Accordion id="myAcc"
        height="60"
        width="200"
        change="handleChange(event);">
        <s:NavigatorContent label="Box 1">
            <s:Label text="This is one view."/>
        </s:NavigatorContent>
        <s:NavigatorContent label="Box 2">
            <s:Label text="This is another view."/>
        </s:NavigatorContent>
    </mx:Accordion>
</s:Application>
```

In this example, you access the `newIndex` property of the IndexChangedEvent object to determine the index of the new child of the Accordion container. For more information on events, see “Events” on page 54.

About the component instantiation life cycle

The component instantiation life cycle describes the sequence of steps that occur when you create a component object from a component class. As part of that life cycle, Flex automatically calls component methods, dispatches events, and makes the component visible.

The following example creates a Button control in ActionScript and adds it to a Group container:

```
// Create a Group container.
var groupContainer:Group = new Group();
// Configure the Group container.
groupContainer.x = 10;
groupContainer.y = 10;

// Create a Button control.
var b:Button = new Button()
// Configure the button control.
b.label = "Submit";
...
// Add the Button control to the Box container.
groupContainer.addElement(b);
```

The following steps show what occurs when you execute the code to create the Button control, and add the control to the container:

1 You call the component's constructor, as the following code shows:

   ```
   // Create a Button control.
   var b:Button = new Button()
   ```

2 You configure the component by setting its properties, as the following code shows:

   ```
   // Configure the button control.
   b.label = "Submit";
   ```

3 You call the `addElement()` method to add the component to its parent, as the following code shows:

   ```
   // Add the Button control to the Box container.
   groupContainer.addElement(b);
   ```

   Flex then performs the following actions:

4 Sets the `parent` property for the component to reference its parent container.

5 Computes the style settings for the component.

6 Dispatches the `preinitialize` event on the component. The component is in a very raw state when this event is dispatched. Many components, such as the Button control, create internal child components to implement functionality. When Flex dispatches the `preinitialize` event, the children (including the internal children, of a component) have not yet been created.

7 Dispatches the `initialize` event on the component. At this time, all of the component's children are initialized, but the component has not been sized or processed for layout. You can use this event to perform additional processing of the component before it is laid out.

   Because the `initialize` event is dispatched early in the component's startup sequence, make sure that none of your processing causes the component to invalidate itself. You typically perform any final processing during the `creationComplete` event.

8 Dispatches the `childAdd` event on the parent container.

9 Dispatches the `initialize` event on the parent container.

10 To display the application, a `render` event gets triggered, and Flex does the following:

   a Flex completes all processing required to display the component, including laying out the component.

   b Makes the component visible by setting the `visible` property to `true`.

   c Dispatches the `creationComplete` event on the component. The component is sized and processed for layout. This event is only dispatched once when the component is created.

d Dispatches the `updateComplete` event on the component. Flex dispatches additional `updateComplete` events whenever the layout, position, size, or other visual characteristic of the component changes and the component is updated for display.

You can later remove a component from a container by using the `removeElement()` method. The removed child's `parent` property is set to `null`. If you add the removed child to another container, it retains its last known state. If there are no references to the component, it is eventually deleted from memory by the garbage collection mechanism of Adobe® Flash® Player.

Given this sequence of actions, you should use the events as follows:

- The `preinitialize` event occurs too early in the component life cycle for most initialization activities. It is useful, however, in the rare situations where you must set the properties on a parent before the children are created.
- To configure a component before Flex has determined its visual appearance, use the `initialize` event. For example, use this for setting properties that affect its appearance, height, or width.
- Use the `creationComplete` event for actions that rely on accurate values for the component's size or position when the component is created. If you use this event to perform an action that changes the visual appearance of the component, Flex must recalculate its layout, which adds unnecessary processing overhead to your application.
- Use the `updateComplete` event for actions that must be performed each time a component's characteristics change, not just when the component is created.

Applying styles

Flex defines styles for setting some of the characteristics of components, such as fonts, padding, and alignment. These are the same styles as those defined and used with Cascading Style Sheets (CSS). Each visual component inherits many of the styles of its superclasses, and can define its own styles. Some styles in a superclass might not be used in a subclass. To determine the styles that a visual component supports, see the styles section of the page for the component in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

You can set all styles in MXML as tag attributes. Therefore, you can set the styles of a BorderContainer container and its contents by using the `borderStyle` and `borderColor` properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\MXMLStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:BorderContainer id="myBC1"
        borderStyle="solid">
        <s:Button label="Submit"/>
    </s:BorderContainer>
    <s:BorderContainer id="myBC2"
        borderStyle="solid"
        borderColor="red">
        <s:Button label="Submit"/>
    </s:BorderContainer>
</s:Application>
```

You can also configure styles in ActionScript by using the `setStyle()` method, or in MXML by using the `<fx:Style>` tag. The `setStyle()` method takes two arguments: the style name and the value. The following example is functionally identical to the previous example:

```
<?xml version="1.0"?>
<!-- components\ComponentsASStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function initBC():void {
                myBC2.setStyle("borderColor", "red");
            }
        ]]>
    </fx:Script>
    <s:BorderContainer id="myBC1"
        borderStyle="solid">
        <mx:Button label="Submit"/>
    </s:BorderContainer>
    <s:BorderContainer id="myBC2"
        borderStyle="solid"
        initialize="initBC();">
        <mx:Button label="Submit"/>
    </s:BorderContainer>
</s:Application>
```

When you use the `<fx:Style>` tag, you set the styles by using CSS syntax or a reference to an external file that contains style declarations, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\TagStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        .myStyle {borderColor: red;}
    </fx:Style>
    <s:BorderContainer id="myBC1"
        borderStyle="solid">
        <mx:Button label="Submit"/>
    </s:BorderContainer>
    <s:BorderContainer id="myBC2"
        borderStyle="solid"
        styleName="myStyle">
        <mx:Button label="Submit"/>
    </s:BorderContainer>
</s:Application>
```

A *class selector* in a style definition, defined as a label preceded by a period, defines a new named style, such as `myStyle` in the preceding example. After you define it, you can apply the style to any component by using the `styleName` property. In the preceding example, you apply the style to the second BorderContainer container.

A *type selector* applies a style to all instances of a particular component type. The following example defines the top and bottom margins for all BorderContainer containers:

```
<?xml version="1.0"?>
<!-- components\TypeSelStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|BorderContainer {borderColor: red;}
    </fx:Style>
    <s:BorderContainer id="myBC1"
        borderStyle="solid">
        <mx:Button label="Submit"/>
    </s:BorderContainer>
    <s:BorderContainer id="myBC2"
        borderStyle="solid">
        <mx:Button label="Submit"/>
    </s:BorderContainer>
</s:Application>
```

In Flex, some, but not all, styles are inherited from parent containers to their children and across style types and classes. Because the `borderStyle` style is not inherited, this example yields results that are identical to the previous examples.

Some Spark and MX components share the same local name. For example, there is a Spark Button component (in the spark.components.* package) and a MX Button component (in the mx.controls.* package). To distinguish between different components that share the same name, you specify namespaces in your CSS that apply to types. The preceding example defines the MX s namespace and uses "`s`" as an identifier:

For more information on styles, see "Styles and themes" on page 1492.

Applying effects

An *effect* is a visible or audible change to the component that occurs over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component.

Effects let you add animation, motion, and sound to your application in response to some user or programmatic action. For example, you can use effects to cause a dialog box to bounce slightly when it receives focus, or to play a sound when the user enters an invalid value.

To create the effect, you define a specific effect with a unique ID in an `<fx:Declarations>` tag. The following example uses two Resize effects for a Button control. One Resize effect expands the size of the button by 10 pixels when the user clicks down on the button, and the second resizes it back to its original size when the user releases the mouse button.

```
<?xml version="1.0"?>
<!-- components\CompIntroBehaviors.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <s:Resize id="myResizeEffectUp"
            target="{myImage}"
            widthBy="10" heightBy="10"/>
        <s:Resize id="myResizeEffectDown"
            target="{myImage}"
            widthBy="-10" heightBy="-10"/>
    </fx:Declarations>
    <mx:Image id="myImage"
        source="@Embed(source='assets/flexlogo.jpg')"/>
    <s:Button label="Resize Me Up"
        click="myResizeEffectUp.end();myResizeEffectUp.play();"/>
    <s:Button label="Resize Me Down"
        click="myResizeEffectDown.end();myResizeEffectDown.play();"/>
</s:Application>
```

For detailed information on using effects, see “Introduction to effects” on page 1784.

Applying skins

Skins are classes that a component uses to control its appearance. These classes are typically written in MXML and are applied to a component by using the skinClass style property. Skin classes can include embedded images or other media, FXG, ActionScript, and any visual component in them.

When creating custom skins, you can use the component’s default skin as a basis for writing your new skin, or the provided Wireframe skins that are simpler.

Skin classes have a contract with a component that enforces certain rules that the skin class and its host component must follow. For example, if a component uses states (such as “mouseOver” or “mouseDown”), the skin class must define the appearance of these states. If a component uses subcomponents (such as a down or up arrow on a slider), the skin class must define these as skin parts.

Typically, a skin class will define a `hostComponent` property so that is can get a reference to the component that uses it. This is not required by the skinning contract but is recommended. Skin classes can also share data with their host component by using data binding.

All components that are subclasses of SkinnableComponent class can have a skin class. Many containers cannot be skinned, however. Only containers that are subclasses of SkinnableContainer can be skinned.

Spark skins can be loaded and unloaded at run time, and their properties can be bound to the style properties of the host component.

For more information on skinning, see “Skinning MX components” on page 1655.

Changing the appearance of a component at run time

You can modify the look, size, or position of a component at run time by using several component properties, styles, or ActionScript methods, including the following:

- `x` and `y`
- `width` and `height`
- styles, by using `setStyle(stylename,value)`

You can set the `x` and `y` properties of a component only when the component is in a container that uses absolute positioning; that is, in a Canvas container, or in an Application or Panel container that has the `layout` property set to `absolute`. All other containers perform automatic layout to set the `x` and `y` properties of their children by using layout rules.

For example, you could use the `x` and `y` properties to reposition a Button control 15 pixels to the right and 15 pixels down in response to a Button control click, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\ButtonMove.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="150"
    height="120">
    <fx:Script>
        <![CDATA[
            public function moveButton():void {
                myButton.x += 15;
                myButton.y += 15;
            }
        ]]>
    </fx:Script>
    <s:Button id="myButton"
        x="15"
        y="15"
        label="Move"
        click="moveButton();"/>
</s:Application>
```

In this application, you can move the Button control without concern for other components. However, moving a component in an application that contains multiple components, or modifying one child of a container that contains multiple children, can cause one component to overlap another, or in some other way affect the layout of the application. Therefore, you should be careful when you perform run-time modifications to container layout.

You can set the `width` and `height` properties for a component in any type of container. The following example increases the `width` and `height` of a Button control by 15 pixels each time the user selects it:

```
<?xml version="1.0"?>
<!-- components\ButtonSize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="150"
    height="150">
    <fx:Script>
        <![CDATA[
            public function resizeButton():void {
                myButton.height = myButton.height + 15;
                myButton.width = myButton.width + 15;
            }
        ]]>
    </fx:Script>
    <s:BorderContainer
        height="80"
        width="100">
        <s:Button id="myButton"
            label="Resize"
            click="resizeButton();"/>
    </s:BorderContainer>
</s:Application>
```

If the container that holds the Button does not use absolute positioning, it repositions its children based on the new size of the Button control.

***Note:** The stored values of `width` and `height` are always in pixels regardless of whether the values were originally set as fixed values, as percentages, or not set at all.*

Extending components

Flex provides several ways for you to extend existing components or to create components. By extending a component, you can add new properties or methods to it.

For example, the following MXML component, defined in the file MyComboBox.mxml, extends the standard Spark ComboBox control to initialize it with the postal abbreviations of the states in New England:

```
<?xml version="1.0"?>
<!-- components\myComponents\MyComboBox.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:dataProvider>
        <mx:ArrayList>
            <fx:String>CT</fx:String>
            <fx:String>MA</fx:String>
            <fx:String>ME</fx:String>
            <fx:String>NH</fx:String>
            <fx:String>RI</fx:String>
            <fx:String>VT</fx:String>
        </mx:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

After you create it, you can use your new component anywhere in your application by specifying its filename as its MXML tag name, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\MainMyComboBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComps="myComponents.*"
    width="150"
    height="150">
    <MyComps:MyComboBox id="stateNames"/>
</s:Application>
```

In this example, the new component is in the myComponents subdirectory. The myComponents.* namespace is mapped to the MyComps identifier.

Flex lets you create custom components by using either of the following methods. The method you choose depends on your application and the requirements of your component:

- Create components as MXML files and use them as custom tags in other MXML files. MXML components provide an easy way to extend an existing component, particularly to modify the behavior of an existing component or add a basic feature to an existing component.
- Create components as ActionScript files by subclassing the UIComponent class or any of its subclasses, and use the resulting classes as custom tags. ActionScript components provide a powerful tool for creating new visual or nonvisual components.

For detailed information on creating custom components, see *"Custom Flex components" on page 2356* .

Data binding

Data binding lets you pass data between client-side objects in an Adobe® Flex™ application. Binding automatically copies the value of a property of a source object to a property of a destination object when the source property changes.

About data binding

Data binding is the process of tying the data in one object to another object. It provides a convenient way to pass data between the different layers of the application. Data binding requires a source property, a destination property, and a triggering event that indicates when to copy the data from the source to the destination. An object dispatches the triggering event when the source property changes.

Adobe Flex provides three ways to specify data binding: the curly braces ({}) syntax in MXML, the <fx:Binding> tag in MXML, and the BindingUtils methods in ActionScript. The following example uses the curly braces ({}) syntax to show a Label control that gets its data from a TextInput control's `text` property:

```
<?xml version="1.0"?>
<!-- binding/BasicBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:TextInput id="myTI" text="Enter text here"/>
    <s:Label id="myText" text="{myTI.text}"/>
</s:Application>
```

The property name inside the curly braces is the source property of the binding expression. When the value of the source property changes, Flex copies the current value of the source property, `myTI.text`, to the destination property, the Label control's `text` property.

You can include ActionScript code and E4X expressions as part of the data binding expressions, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/BasicBindingWithAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:TextInput id="myTI"/>
    <s:Label id="myText" text="{myTI.text.toUpperCase()}"/>
</s:Application>
```

In this example, you use the ActionScript method `String.toUpperCase()` to convert the text in the source property to upper case when Flex copies it to the destination property. For more information, see "Using ActionScript in data binding expressions" on page 313 and "Using an E4X expression in a data binding expression" on page 316.

You can use the <fx:Binding> tag as an alternative to the curly braces syntax. When you use the `<fx:Binding>` tag, you provide a source property in the `<fx:Binding>` tag's `source` property and a destination property in its `destination` property. The following example uses the `<fx:Binding>` tag to define a data binding from a TextInput control to a Label control:

```
<?xml version="1.0"?>
<!-- binding/BasicBindingMXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Binding source="myTI.text" destination="myText.text"/>
    <s:TextInput id="myTI"/>
    <s:Label id="myText"/>
</s:Application>
```

In contrast to the curly braces syntax, you can use the `<fx:Binding>` tag to completely separate the view (user interface) from the model. The `<fx:Binding>` tag also lets you bind multiple source properties to the same destination property because you can specify multiple `<fx:Binding>` tags with the same destination. For an example, see "Binding more than one source property to a destination property" on page 305.

The curly braces syntax and the `<fx:Binding>` tag both define a data binding at compile time. You can also use ActionScript code to define a data binding at run time, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/BasicBindingAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
      <![CDATA[
        import mx.binding.utils.*;
        // Define data binding.
        public function initBindingHandler():void {
            BindingUtils.bindProperty(myText, "text", myTI, "text");
        }
      ]]>
    </fx:Script>
    <s:TextInput id="myTI"/>
    <s:Label id="myText" preinitialize="initBindingHandler();"/>
</s:Application>
```

In this example, you use the static `BindingUtils.bindProperty()` method to define the binding. You can also use the `BindingUtils.bindSetter()` method to define a binding to a function. For more information, see “Defining data bindings in ActionScript” on page 318.

Notice in this example that you use the `preinitialize` event to define the data binding. This is necessary because Flex triggers all data bindings at application startup when the source object dispatches the `initialize` event. For more information, see “When data binding occurs” on page 301.

When data binding occurs

Binding occurs under the following circumstances:

1 The binding source dispatches an event because the source has been modified.

   This event can occur at any time during application execution. The event triggers Flex to copy the value of the source property to the destination property.

2 At application startup when the source object dispatches the `initialize` event.

   All data bindings are triggered once at application startup to initialize the destination property.

To monitor data binding, you can define a binding watcher that triggers an event handler when a data binding occurs. For more information, see “Defining binding watchers” on page 320.

The `executeBindings()` method of the UIComponent class executes all the bindings for which a UIComponent object is the destination. All containers and controls, as well as the Repeater component, extend the UIComponent class. The `executeChildBindings()` method of the Container and Repeater classes executes all of the bindings for which the child UIComponent components of a Container or Repeater class are destinations. All containers extend the Container class.

These methods give you a way to execute bindings that do not occur as expected. By adding one line of code, such as a call to the `executeChildBindings()` method, you can update the user interface after making a change that does not cause bindings to execute. However, you should only use the `executeBindings()` method when you are sure that bindings do not execute automatically.

Properties that support data binding

You can use all properties of an object as the destination of a data binding expression. However, to use a property as the source of a data binding expression, the source object must be implemented to support data binding, which means that the object dispatches an event when the value of the property changes to trigger the binding. In this topic, a property that can be used as the source of a data-binding expression is referred to as a *bindable* property.

In the *ActionScript 3.0 Reference for the Adobe Flash Platform*, a property that can be used as the source of a data binding expression includes the following statement in its description:

“This property can be used as the source for data binding.”

Using read-only properties as the source for data binding

You can use a read-only property defined by a getter method, which means no setter method, as the source for a data-binding expression. Flex performs the data binding once when the application starts.

Using static properties as the source for data binding

You can automatically use a static constant as the source for a data-binding expression. Flex performs the data binding once when the application starts.

You can use a static variable as the source for a data-binding expression. Flex performs the data binding once when the application starts.

Creating properties to use as the source for data binding

When you create a property that you want to use as the source of a data binding expression, Flex can automatically copy the value of the source property to any destination property when the source property changes. To signal to Flex to perform the copy, you must use the `[Bindable]` data tag to register the property with Flex.

The `[Bindable]` metadata tag has the following syntax:

```
[Bindable]
[Bindable(event="eventname")]
```

You can optionally omit the `event` keyword, and specify the tag as shown below:

```
[Bindable("eventname")]
```

If you omit the event name, Flex automatically creates an event named `propertyChange`, and Flex dispatches that event when the property changes to trigger any data bindings that use the property as a data-binding source. If you specify the event name, it is your responsibility to dispatch the event when the source property changes. For more information and examples of using the `[Bindable]` metadata tag, see “Using the Bindable metadata tag” on page 320.

The following example makes the `maxFontSize` and `minFontSize` properties that you defined as variables usable as the sources for data bindings expressions:

```
<?xml version="1.0"?>
<!-- binding/FontPropertyBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            // Define public vars for tracking font size.
            [Bindable]
            public var maxFontSize:Number = 15;
            [Bindable]
            public var minFontSize:Number = 5;
        ]]>
    </fx:Script>

    <s:Label text="{maxFontSize}"/>
    <s:Label text="{minFontSize}"/>
    <s:Button click="maxFontSize=20; minFontSize=10;"/>
</s:Application>
```

When you click the Button control, you update the values of the `maxFontSize` and `minFontSize` properties, and trigger a data binding update to the Label controls.

***Note:** If you omit the `[Bindable]` metadata tag, the compiler issues a warning stating that the data binding mechanism cannot detect changes to the property. For more information, see "Using the Bindable metadata tag" on page 320.*

Data binding uses

Common uses of data binding include the following:

- To bind properties of user interface controls to other user interface controls.
- To bind properties of user interface controls to a middle-tier data model, and to bind that data model's fields bound to a data service request (a three-tier system).
- To bind properties of user interface controls to data service requests.
- To bind data service results to properties of user interface controls.
- To bind data service results to a middle-tier data model, and to bind that data model's fields to user interface controls. For more information about data models, see "Storing data" on page 889.
- To bind an ArrayCollection or XMLListCollection object to the `dataProvider` property of a List-based control.
- To bind individual parts of complex properties to properties of user interface controls. An example would be a master-detail scenario in which clicking an item in a List control displays data in several other controls.
- To bind XML data to user interface controls by using ECMAScript for XML (E4X) expressions in binding expressions.

Although binding is a powerful mechanism, it is not appropriate for all situations. For example, for a complex user interface in which individual pieces must be updated based on strict timing, it would be preferable to use a method that assigns properties in order. Also, binding executes every time a property changes, so it is not the best solution when you want changes to be noticed only some of the time.

Data binding examples

Using data binding with data models

In the following example, a set of UI control properties act as the binding source for a data model. For more information about data models, see “Storing data” on page 889.

```
<?xml version="1.0"?>
<!-- binding/BindingBraces.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Data model stores registration data that user enters. -->
        <fx:Model id="reg">
            <registration>
                <name>{fullname.text}</name>
                <email>{email.text}</email>
                <phone>{phone.text}</phone>
                <zip>{zip.text}</zip>
                <ssn>{ssn.text}</ssn>
            </registration>
        </fx:Model>
    </fx:Declarations>
    <!-- Form contains user input controls. -->
    <s:Form>
        <s:FormItem label="Name" required="true">
            <s:TextInput id="fullname" width="200"/>
        </s:FormItem>
        <s:FormItem label="Email" required="true">
            <s:TextInput id="email" width="200"/>
        </s:FormItem>
        <s:FormItem label="Phone" required="true">
            <s:TextInput id="phone" width="200"/>
        </s:FormItem>
        <s:FormItem label="Zip" required="true">
            <s:TextInput id="zip" width="60"/>
        </s:FormItem>
        <s:FormItem label="Social Security" required="true">
            <s:TextInput id="ssn" width="200"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Binding a source property to more than one destination property

You can bind a single source property to more than one destination property by using the curly braces ({}) syntax, the <fx:Binding> tag, or the `BindingUtils`methods in ActionScript. In the following example, a TextInput control’s `text` property is bound to properties of two data models, and the data model properties are bound to the `text` properties of two Label controls.

```
<?xml version="1.0"?>
<!-- binding/BindMultDestinations.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:Model id="mod1">
            <data>
                <part>{input1.text}</part>
            </data>
        </fx:Model>
        <fx:Model id="mod2">
            <data>
                <part>{input1.text}</part>
            </data>
        </fx:Model>
    </fx:Declarations>
    <s:TextInput id="input1" text="Hello" />
    <s:Label text="{mod1.part}"/>
    <s:Label text="{mod2.part}"/>
</s:Application>
```

Binding more than one source property to a destination property

You can bind more than one source property to the same destination property. You can set up one of these bindings by using curly braces, but you must set up the others by using the `<fx:Binding>` tag, or by using calls to the `BindingUtils.bindProperty()` method or to the `BindingUtils.bindSetter()` method.

In the following example, the TextArea control is the binding destination, and both `input1.text` and `input2.text` are its binding sources:

```
<?xml version="1.0"?>
<!-- binding/BindMultSources.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Binding source="input2.text" destination="myTA.text"/>
    <s:TextInput id="input1"/>
    <s:TextInput id="input2"/>

    <s:TextArea id="myTA" text="{input1.text}"/>
</s:Application>
```

If `input1.text` or `input2.text` is updated, the TextArea control contains the updated value.

Defining bidirectional bindings

Bidirectional, or two-way, data binding occurs when two objects act as the source and the destination for each other. When you modify the source property of either object, the destination property of the other object is updated.

Define a bidirectional data binding using one of the following methods:

1 Define two objects that specify as the source a property of the other object. In the following example, input1 specifies input2.text as the source property, and input2 specifies input1.text as the source property. Any change to input1.text updates input2.text, and any change to input2.text updates input1.text:

```
<!-- Specify data binding for both controls. -->
<s:TextInput id="input1" text="{input2.text}"/>
<s:TextInput id="input2" text="{input1.text}"/>
```

2 Use the `@{bindable_property}` syntax for one source property, as the following example shows:

```
<!-- Specify data binding for both controls. -->
<s:TextInput id="input1" text="@{input2.text}"/>
<s:TextInput id="input2"/>
```

***Note:** The property definition that includes the `@{bindable_property}` syntax is called the primary property. If the primary property has not had a value assigned to it, the binding to it occurs first, before a binding to the other property.*

3 Use the `twoWay` property of the `<fx:Binding>` tag, as the following example shows:

```
<fx:Binding source="input1.text" destination="input2.text" twoWay="true"/>
```

Because both the source and the destination properties must resolve to a bindable property or property chain at compile time, neither property value can include a binding expression.

The following example shows these three techniques:

```
<?xml version="1.0"?>
<!-- binding/BindBiDirection.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="10"/>
    </s:layout>
    <s:Label text="Specify data binding for both controls."/>
    <s:TextInput id="input1" text="{input2.text}"/>
    <s:TextInput id="input2" text="{input1.text}"/>
    <s:Label text="Use the bidirectional data binding syntax."/>
    <s:TextInput id="input3" text="@{input4.text}"/>
    <s:TextInput id="input4"/>

    <s:Label text="Use the Binding tag."/>
    <s:TextInput id="input5"/>
    <s:TextInput id="input6"/>
    <fx:Binding source="input5.text" destination="input6.text" twoWay="true"/>
</s:Application>
```

In this example, the TextInput controls are the source of a data binding expression, and both are a destination. For example, when you modify input1, its value is copied to input2, and when you modify input2, its value is copied to input1.

Flex ensures that bidirectional data bindings do not result in an infinite loop; that is, Flex ensures that a bidirectional data binding is triggered only once when either source property is modified.

Restrictions on where you can use bidirectional data binding

Bidirectional data bindable is supported in most places in an application. However, the following table shows where bidirectional data binding is not supported:

Expression	Bidirectional Binding Supported
Style properties	No
Effect properties	No
The `request` property of the HttpService, RemoteObject, and WebService classes.	No
The `arguments` property of the RemoteObject class.	No

Binding compatibility with Flex 3

In Flex 3, the following binding expression was interpreted as a one-way binding:

```
<mx:String id="str">test</mx:String>
<mx:Text text="@{str}"/>
```

Flex 3 set the Text component’s `text` property to “test”, but the value of the String component did not change if the Text component’s `text` property changed.

In Flex 4 and later, this code is interpreted as a bidirectional data binding expression. A change in the String component updates the `text` property of the Text component, and a change in the `text` property of the Text component updates the String component. You can prevent the compiler from enforcing the bidirectional binding by setting the value of the `compatibility-version` compiler option to `3.0.0`, as the following example shows:

```
mxmlc ... -compatibility-version=3.0.0 ...
```

For more information, see “Backward compatibility” on page 2239.

Binding to functions, Objects, and Arrays

Using functions as the source for a data binding

You can use a function as part of the source of a data binding expression. Two common techniques with functions are to use bindable properties as arguments to the function to trigger the function, or to trigger the function in response to a binding event.

Using functions that take bindable properties as arguments

You can use ActionScript functions as the source of data binding expressions when using a bindable property as an argument of the function. When the bindable property changes, the function executes, and the result is written to the destination property, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/ASInBraces.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:CurrencyFormatter id="usdFormatter" useCurrencySymbol="true"/>
    </fx:Declarations>
    <s:TextInput id="myTI" text="Enter number here"/>
    <s:TextArea text="{usdFormatter.format(myTI.text)}"/>
</s:Application>
```

In this example, Flex calls the `CurrencyFormatter.format()` method to update the TextArea control every time the `text` property of the TextInput control is modified.

If the function is not passed an argument that can be used as the source of a data binding expression, the function only gets called once when the applications starts.

Binding to functions in response to a data-binding event

You can specify a function that takes no bindable arguments as the source of a data binding expression. However, you then need a way to invoke the function to update the destination of the data binding.

In the following example, you use the `[Bindable]` metadata tag to specify to Flex to invoke the `isEnabled()` function in response to the event `myFlagChanged`. When the myFlag setter gets called, it dispatches the `myFlagChanged` event to trigger any data bindings that use the `isEnabled()` function as the source:

```
<?xml version="1.0"?>
<!-- binding/ASFunction.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;

            // Define a function that gets invoked
            // in response to the myFlagChanged event.
            [Bindable(event="myFlagChanged")]
            private function isEnabled():String {
                if (myFlag)
                    return 'true';
                else
                    return 'false';
            }
            private var _myFlag:Boolean = false;
            // Define a setter method that dispatches the
            // myFlagChanged event to trigger the data binding.
            public function set myFlag(value:Boolean):void {
                _myFlag = value;
                dispatchEvent(new Event("myFlagChanged"));
            }
            public function get myFlag():Boolean {
                return _myFlag;
            }
        ]]>
    </fx:Script>
    <!-- Use the function as the source of a data binding expression. -->
    <s:TextArea id="myTA" text="{isEnabled()}"/>
    <!-- Modify the property, causing the setter method to
        dispatch the myFlagChanged event to trigger data binding. -->
    <s:Button label="Clear MyFlag" click="myFlag=false;"/>
    <s:Button label="Set MyFlag" click="myFlag=true;"/>
</s:Application>
```

For more information on the `[Bindable]` metadata tag, see “Using the Bindable metadata tag” on page 320.

Using data binding with Objects

When working with Objects, you have to consider when you define a binding to the Object, or when you define a binding to a property of the Object.

Binding to Objects

When you make an Object the source of a data binding expression, the data binding occurs when the Object is updated, or when a reference to the Object is updated, but not when an individual field of the Object is updated.

In the following example, you create subclass of Object that defines two properties, `stringProp` and `intProp`, but does not make the properties bindable:

```
package myComponents
{
    // binding/myComponents/NonBindableObject.as
    // Make no class properties bindable.
    public class NonBindableObject extends Object {

        public function NonBindableObject() {
            super();
        }

        public var stringProp:String = "String property";
        public var intProp:int = 52;
    }
}
```

Since the properties of the class are not bindable, Flex will not dispatch an event when they are updated to trigger data binding. You then use this class in a Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/WholeObjectBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initObj();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import myComponents.NonBindableObject;
            [Bindable]
            public var myObj:NonBindableObject = new NonBindableObject();
            [Bindable]
            public var anotherObj:NonBindableObject =
                new NonBindableObject();
            public function initObj():void {
                anotherObj.stringProp = 'anotherObject';
                anotherObj.intProp = 8;
            }
        ]]>
    </fx:Script>
    <!-- Data binding updated at application startup. -->
    <s:Label id="text1" text="{myObj.stringProp}"/>
    <!-- Data binding updated at application startup. -->
    <s:Label id="text2" text="{myObj.intProp}"/>
    <!-- Data bindings to stringProp not updated. -->
    <s:Button label="Change myObj.stringProp"
        click="myObj.stringProp = 'new string';"/>

    <!-- Data bindings to intProp not updated. -->
    <s:Button label="Change myObj.intProp"
        click="myObj.intProp = 10;"/>

    <!-- Data bindings to myObj and to myObj properties updated. -->
    <s:Button label="Change myObj"
        click="myObj = anotherObj;"/>
</s:Application>
```

Because you did not make the individual fields of the NonBindableObject class bindable, the data bindings for the two Text controls are updated at application start up, and when myObj is updated, but not when the individual properties of myObj are updated.

When you compile the application, the compiler outputs warning messages stating that the data binding mechanism will not be able to detect changes to stringProp and intProp.

Binding to properties of Objects

To make properties of an Object bindable, you create a new class definition for the Object, as the following example shows:

```
package myComponents
{
    // binding/myComponents/BindableObject.as
    // Make all class properties bindable.
    [Bindable]
    public class BindableObject extends Object {

        public function BindableObject() {
            super();
        }

        public var stringProp:String = "String property";
        public var intProp:int = 52;
    }
}
```

By placing the `[Bindable]` metadata tag before the class definition, you make bindable all public properties defined as variables, and all public properties defined by using both a setter and a getter method. You can then use the stringProp and intProp properties as the source for a data binding, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/SimpleObjectBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initObj();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import myComponents.BindableObject;
            [Bindable]
            public var myObj:BindableObject = new BindableObject();
            [Bindable]
            public var anotherObj:BindableObject =
                new BindableObject();
            public function initObj():void {
                anotherObj.stringProp = 'anotherObject';
                anotherObj.intProp = 8;
            }
        ]]>
    </fx:Script>
    <!-- Data binding updated at application startup. -->
    <s:Label id="text1" text="{myObj.stringProp}"/>
    <!-- Data binding updated at application startup. -->
    <s:Label id="text2" text="{myObj.intProp}"/>
    <!-- Data bindings to stringProp updated. -->
    <s:Button label="Change myObj.stringProp"
        click="myObj.stringProp = 'new string';"/>

    <!-- Data bindings to intProp updated. -->
    <s:Button label="Change myObj.intProp"
        click="myObj.intProp = 10;"/>

    <!-- Data bindings to myObj and to myObj properties updated. -->
    <s:Button label="Change myObj"
        click="myObj = anotherObj;"/>
</s:Application>
```

Binding with arrays

When working with arrays, such as Array or ArrayCollection objects, you can define the array as the source or destination of a data binding expression.

***Note:** When defining a data binding expression that uses an array as the source of a data binding expression, the array should be of type ArrayCollection because the ArrayCollection class dispatches an event when the array or the array elements change to trigger data binding. For example, a call to `ArrayCollection.addItem()`, `ArrayCollection.addItemAt()`, `ArrayCollection.removeItem()`, and `ArrayCollection.removeItemAt()` all trigger data binding.*

Binding to arrays

You often bind arrays to the `dataProvider` property of Flex controls, as the following example shows for the List control:

```
<?xml version="1.0"?>
<!-- binding/ArrayBindingDP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            public var myAC:ArrayCollection = new ArrayCollection([
                "One", "Two", "Three", "Four"]);

            [Bindable]
            public var myAC2:ArrayCollection = new ArrayCollection([
                "Uno", "Dos", "Tres", "Quatro"]);
        ]]>
    </fx:Script>

    <!-- Data binding updated at application startup,
        when myAC is modified, and when an element of
        myAC is modifed. -->
    <s:List width="150"
        dataProvider="{myAC}"/>
    <!-- Data bindings to myAC updated. -->
    <s:Button
        label="Change Element"
        click="myAC[0]='mod One'"/>

    <!-- Data bindings to myAC updated. -->
    <s:Button
        label="Add Element"
        click="myAC.addItem('new element');"/>

    <!-- Data bindings to myAC updated. -->
    <s:Button
        label="Remove Element 0"
        click="myAC.removeItemAt(0);"/>

    <!-- Data bindings to myAC updated. -->
    <s:Button
        label="Change ArrayCollection"
        click="myAC=myAC2"/>
</s:Application>
```

This example defines an ArrayCollection object, and then uses data binding to set the data provider of the List control to the ArrayCollection. When you modify an element of the ArrayCollection object, or when you modify a reference to the ArrayCollection object, you trigger a data binding.

Binding to array elements

You can use individual ArrayCollection elements as the source or a binding expression, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/ArrayBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            public var myAC:ArrayCollection = new ArrayCollection([
                "One", "Two", "Three", "Four"]);

            [Bindable]
            public var myAC2:ArrayCollection = new ArrayCollection([
                "Uno", "Dos", "Tres", "Quatro"]);
        ]]>
    </fx:Script>
    <!-- Data binding updated at application startup
        and when myAC modified. -->
    <s:Label id="text1" text="{myAC[0]}"/>
    <!-- Data binding updated at application startup,
        when myAC modified, and when myAC[0] modified. -->
    <s:Label id="text2" text="{myAC.getItemAt(0)}"/>

    <s:Button id="button1"
        label="Change Element"
        click="myAC[0]='new One'"/>
    <s:Button id="button2"
        label="Change ArrayCollection"
        click="myAC=myAC2"/>
</s:Application>
```

If you specify an array element as the source of a data binding expression by using the square bracket syntax, [], data binding is only triggered when the application starts and when the array or a reference to the array is updated; data binding is not triggered when the individual array element is updated. The compiler issues a warning in this situation.

However, the data binding expression `myAC.getItemAt(0)` is triggered when an array element changes. Therefore, the text2 Text control is updated when you click button1, while text1 is not. When using an array element as the source of a data binding expression, you should use the `ArrayCollection.getItemAt()` method in the binding expression.

Clicking button2 copies myAC2 to myAC, and triggers all data bindings to array elements regardless of how you implemented them.

Using ActionScript in data binding expressions

You can use ActionScript in data binding expressions defined by curly braces and by the `<fx:Binding>` tag; however, you cannot use ActionScript when defining a data binding by using the `BindingUtils.bindProperty()` or the `BindingUtils.bindSetter()` method.

Using ActionScript expressions in curly braces

Binding expressions in curly braces can contain an ActionScript expression that returns a value. For example, you can use the curly braces syntax for the following types of binding.

- A single bindable property inside curly braces
- To cast the data type of the source property to a type that matches the destination property
- String concatenation that includes a bindable property inside curly braces
- Calculations on a bindable property inside curly braces
- Conditional operations that evaluate a bindable property value

The following example shows a data model that uses each type of binding expression:

```
<?xml version="1.0"?>
<!-- binding/AsInBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:Model id="myModel">
            <myModel>
              <!-- Perform simple property binding. -->
              <a>{nameInput.text}</a>
              <!-- Perform string concatenation. -->
              <b>This is {nameInput.text}</b>
              <!-- Perform a calculation. -->
              <c>{(Number(numberInput.text)) * 6 / 7}</c>
              <!-- Perform a conditional operation using a ternary operator. -->
              <d>{(isMale.selected) ? "Mr." : "Ms."} {nameInput.text}</d>
            </myModel>
        </fx:Model>
    </fx:Declarations>
    <s:Form>
        <s:FormItem label="Last Name:">
```

```
            <s:TextInput id="nameInput"/>
        </s:FormItem>
        <s:FormItem label="Select sex:">
            <s:RadioButton id="isMale"
                label="Male"
                groupName="gender"
                selected="true"/>
            <s:RadioButton id="isFemale"
                label="Female"
                groupName="gender"/>
        </s:FormItem>
        <s:FormItem label="Enter a number:">
            <s:TextInput id="numberInput" text="0"/>
        </s:FormItem>
    </s:Form>

    <s:Label
        text="{'Calculation: '+numberInput.text+' * 6 / 7 = '+myModel.c}"/>
    <s:Label text="{'Conditional: '+myModel.d}"/>
</s:Application>
```

Using ActionScript expressions in Binding tags

The `source` property of an `<fx:Binding>` tag can contain curly braces. When there are no curly braces in the `source` property, the value is treated as a single ActionScript expression. When there are curly braces in the `source` property, the value is treated as a concatenated ActionScript expression. The `<fx:Binding>` tags in the following example are valid and equivalent to each other:

```
<?xml version="1.0"?>
<!-- binding/ASInBindingTags.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function whatDogAte():String {
                return "homework";
            }
        ]]>
    </fx:Script>
    <fx:Binding
        source="'The dog ate my '+ whatDogAte()"
        destination="field1.text"/>
    <fx:Binding
        source="{'The dog ate my '+ whatDogAte()}"
        destination="field2.text"/>
    <fx:Binding
        source="The dog ate my {whatDogAte()}"
        destination="field3.text"/>
    <s:TextArea id="field1"/>
    <s:TextArea id="field2"/>
    <s:TextArea id="field3"/>
</s:Application>
```

The `source` property in the following example is not valid because it is not an ActionScript expression:

```
<fx:Binding source="The dog ate my homework" destination="field1.text"/>
```

Using an ampersand character in a data binding expression

Because of the parsing rules of XML, if you want to use an ampersand character, `&`, in a data binding expression in an MXML file, you must replace it with the hexadecimal equivalent character, `&amp;`. For example, if you want to use a logical AND expression, written in ActionScript as `&&`, then you have to write is as `&amp;&amp;`, as the following example shows:

```
<s:Button label="Test" enabled="{authorized &amp;&amp; cc}" />
```

Using an E4X expression in a data binding expression

A binding expression in curly braces or an `<fx:Binding>` tag can contain an ECMAScript for XML (E4X) expression when the source of a binding is a bindable property of type XML. You cannot use E4X when defining a data binding by using the `BindingUtils.bindProperty()` or the `BindingUtils.bindSetter()` method.

Using an E4X expression in curly braces

A binding expression in curly braces automatically calls the `toString()` method when the binding destination is a String property. A binding expression in curly braces or an `<fx:Binding>` tag can contain an ECMAScript for XML (E4X) expression when the source of a binding is a bindable property of type XML; for more information, see “Using an E4X expression in an <fx:Binding> tag” on page 317.

In the code in the following example, there are three binding expressions in curly braces that bind data from an XML object. The first uses . (dot) notation, the second uses .. (dot dot) notation, and the third uses || (or) notation.

```
<?xml version="1.0"?>
<!-- binding/E4XInBraces.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="750">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            [Bindable]
            public var xdata:XML = <order>
                <item id = "3456">
                    <description>Big Screen Television</description>
                    <price>1299.99</price><quantity>1</quantity>
                </item>
                <item id = "56789">
                    <description>DVD Player</description>
                    <price>399.99</price>
                    <quantity>1</quantity>
                </item>
            </order>;
        ]]>
    </fx:Script>
    <mx:Label text="Using .. notation."/>
    <!-- Inline databinding will automatically call the
        toString() method when the binding destination is a string. -->
    <mx:List width="25%"
        dataProvider="{xdata..description}"/>

    <mx:Label text="Using . notation."/>
    <mx:List width="25%"
        dataProvider="{xdata.item.description}"/>
    <mx:Label text="Using || (or) notation."/>
    <mx:List width="25%"
        dataProvider="{xdata.item.(@id=='3456'||@id=='56789').description}"/>
</s:Application>
```

Using an E4X expression in an <fx:Binding> tag

Unlike an E4X expression in curly braces, when you use an E4X expression in an `<fx:Binding>` tag, you must explicitly call the `toString()` method when the binding destination is a String property.

In the code in the following example, there are three binding expressions in curly braces that bind data from an XML object. The first uses . (dot) notation, the second uses .. (dot dot) notation, and the third uses || (or) notation.

```
<?xml version="1.0"?>
<!-- binding/E4XInBindingTag.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="600" height="900">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var xdata:XML =
                <order>
                    <item id = "3456">
                        <description>Big Screen Television</description>
                        <price>1299.99</price><quantity>1</quantity>
                    </item>
                    <item id = "56789">
                        <description>DVD Player</description>
                        <price>399.99</price>
                        <quantity>1</quantity>
                    </item>
                </order>;
        ]]>
    </fx:Script>
    <mx:Label text="Using .. notation."/>
    <!-- This will update because what is
        binded is actually the String and XMLList. -->
    <mx:List width="75%" id="txts"/>
    <mx:Label text="Using . notation."/>
    <mx:List width="75%" id="txt2s"/>
    <mx:Label text="Using || (or) notation."/>
    <mx:List width="75%" id="txt3s"/>
    <fx:Binding
        source="xdata..description"
        destination="txts.dataProvider"/>
    <fx:Binding
        source="xdata.item.description"
        destination="txt2s.dataProvider"/>
    <fx:Binding
        source="xdata.item.(@id=='3456'||@id=='56789').description"
        destination="txt3s.dataProvider"/>
</s:Application>
```

Defining data bindings in ActionScript

You can define a data binding in ActionScript by using the mx.binding.utils.BindingUtils class. This class defines static methods that let you create a data binding to a property implemented as a variable, by using the `bindProperty()` method, or to a method, by using the `bindSetter()` method. For an example using the `bindProperty()` method, see “About data binding” on page 299.

Differences between defining bindings in MXML and ActionScript

There are a few differences between defining data bindings in MXML at compile time and in defining them at runtime in ActionScript:

- You cannot include ActionScript code in a data binding expression defined by the `bindProperty()` or `bindSetter()` method. Instead, use the `bindSetter()` method to specify a method to call when the binding occurs.
- You cannot include an E4X expression in a data binding expression defined in ActionScript.
- You cannot include functions or array elements in property chains in a data binding expression defined by the `bindProperty()` or `bindSetter()` method.
- The MXML compiler has better warning and error detection support than runtime data bindings defined by the `bindProperty()` or `bindSetter()` method.

Example: Defining a data binding in ActionScript

The following example uses the `bindSetter()` method to set up a data binding. The arguments to the `bindSetter()` method specify the following:

- The source object
- The name of the source property
- A method that is called when the source property changes

In the following example, as you enter text in the TextInput control, the text is converted to upper case as it is copied to the TextArea control:

```
<?xml version="1.0"?>
<!-- binding/BindSetterAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.binding.utils.*;
            import mx.events.FlexEvent;
            // Method called when myTI.text changes.
            public function updateMyString(val:String):void {
                myTA.text = val.toUpperCase();
            }
            <!-- Event listener to configure binding. -->
            public function mySetterBinding(event:FlexEvent):void {
                var watcherSetter:ChangeWatcher =
                    BindingUtils.bindSetter(updateMyString, myTI, "text");
            }
        ]]>
    </fx:Script>
    <s:Label text="Bind Setter using setter method"/>
    <s:TextInput id="myTI"
        text="Hello Setter" />
    <s:TextArea id="myTA"
        initialize="mySetterBinding(event);"/>
</s:Application>
```

Defining binding watchers

Flex includes the mx.binding.utils.ChangeWatcher class that you can use to define a data-binding watcher. Typically, a data-binding watcher invokes an event listener when a binding occurs. To set up a data-binding watcher, you use the static `watch()` method of the ChangeWatcher class, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/DetectWatcher.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initWatcher();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
          import mx.binding.utils.*;
          import mx.events.FlexEvent;
          import mx.events.PropertyChangeEvent;
          public var myWatcher:ChangeWatcher;
          // Define binding watcher.
          public function initWatcher():void {
            // Define a watcher for the text binding.
              myWatcher = ChangeWatcher.watch(textarea, "text", watcherListener);
          }
          // Event listener when binding occurs.
          public function watcherListener(event:Event):void {
            myTA1.text="binding occurred";

            // Use myWatcher.unwatch() to remove the watcher.
          }
        ]]>
    </fx:Script>
    <!-- Define a binding expression to watch. -->
    <s:TextInput id="textinput" text="Hello"/>
    <s:TextArea id="textarea" text="{textinput.text}"/>
    <!-- Trigger a binding. -->
    <s:Button label="Submit" click="textinput.text='Goodbye';"/>
    <s:TextArea id="myTA1"/>
</s:Application>
```

You define an event listener for the data-binding watcher, where the event listener takes a single argument that contains the event object. The data type of the event object is determined by the property being watched. Each bindable property can dispatch a different event type and the associated event object. For more information on determining the event type, see “Using the Bindable metadata tag” on page 320.

Many event listeners take an event object of type PropertyChangeEvent because that is the default data type of the event dispatched by bindable properties. You can always specify an event type of flash.events.Event, the base class for all Flex events.

Using the Bindable metadata tag

When a property is the source of a data binding expression, Flex automatically copies the value of the source property to any destination property when the source property changes. To signal to Flex to perform the copy, you must use the `[Bindable]` metadata tag to register the property with Flex, and the source property must dispatch an event.

The `[Bindable]` metadata tag has the following syntax:

```
[Bindable]
[Bindable(event="eventname")]
```

If you omit the event name, Flex automatically creates an event named `propertyChange` of type PropertyChangeEvent.

You can use the `[Bindable]` metadata tag in three places:

1 Before a public class definition.

   The `[Bindable]` metadata tag makes usable as the source of a binding expression all public properties that you defined as variables, and all public properties that are defined by using both a setter and a getter method. In this case, `[Bindable]` takes no parameters, as the following example shows:

   ```
   [Bindable]
   public class TextAreaFontControl extends TextArea {}
   ```

   The compiler automatically generates an event named `propertyChange`, of type PropertyChangeEvent, for all public properties so that the properties can be used as the source of a data binding expression.

   If the property value remains the same on a write, Flex does not dispatch the event or update the property, where *not the same* translates to the following test:

   ```
   (oldValue !== value)
   ```

   That means if a property contains a reference to an object, and that reference is modified to reference a different but equivalent object, the binding is triggered. If the property is not modified, but the object that it points to changes internally, the binding is not triggered.

   ***Note:** When you use the `[Bindable]` metadata tag before a public class definition, it only applies to public properties; it does not apply to private or protected properties, or to properties defined in any other namespace. You must insert the `[Bindable]` metadata tag before a nonpublic property to make it usable as the source for a data binding expression.*

2 Before a public, protected, or private property defined as a variable to make that specific property support binding.

   The tag can have the following forms:

   ```
   [Bindable]
   public var foo:String;
   ```

   The Flex compiler automatically generates an event named `propertyChange`, of type PropertyChangeEvent, for the property. If the property value remains the same on a write, Flex does not dispatch the event or update the property.

   You can also specify the event name, as the following example shows:

   ```
   [Bindable(event="fooChanged")]
   public var foo:String;
   ```

   In this case, you are responsible for generating and dispatching the event, typically as part of some other method of your class. You can specify a `[Bindable]` tag that includes the `event` specification if you want to name the event, even when you already specified the `[Bindable]` tag at the class level.

3 Before a public, protected, or private property defined by a getter or setter method.

You must define both a setter and a getter method to use the `[Bindable]` tag with the property. If you define just a setter method, you create a write-only property that you cannot use as the source of a data-binding expression. If you define just a getter method, you create a read-only property that you can use as the source of a data-binding expression without inserting the `[Bindable]` metadata tag. This is similar to the way that you can use a variable, defined by using the `const` keyword, as the source for a data binding expression.

The tag can have the following forms:

```
[Bindable]
public function set shortNames(val:Boolean):void {
    ...
}

public function get shortNames():Boolean {
    ...
}
```

The Flex compiler automatically generates an event named `propertyChange`, of type PropertyChangeEvent, for the property. If the property value remains the same on a write, Flex does not dispatch the event or update the property. To determine if the property value changes, Flex calls the getter method to obtain the current value of the property.

You can specify the event name, as the following example shows:

```
[Bindable(event="changeShortNames")]
public function set shortNames(val:Boolean):void {
    ...
    // Create and dispatch event.
    dispatchEvent(new Event("changeShortNames"));
}

// Get method.
public function get shortNames():Boolean {
    ...
}
```

In this case, you are responsible for generating and dispatching the event, typically in the setter method, and Flex does not check to see if the old value and the new value are different. You can specify a `[Bindable]` tag that includes the `event` specification to name the event, even when you already specified the `[Bindable]` tag at the class level.

The following example makes the `maxFontSize` and `minFontSize` properties that you defined as variables that can be used as the sources for data bindings:

```
 // Define public vars for tracking font size.
 [Bindable]
public var maxFontSize:Number = 15;
 [Bindable]
public var minFontSize:Number = 5;
```

In the following example, you make a public property that you defined by using a setter and a getter method that is usable as the source for data binding The `[Bindable]` metadata tag includes the name of the event broadcast by the setter method when the property changes:

```
    // Define private variable.
    private var _maxFontSize:Number = 15;

    [Bindable(event="maxFontSizeChanged")]
    // Define public getter method.
    public function get maxFontSize():Number {
        return _maxFontSize;
    }

    // Define public setter method.
    public function set maxFontSize(value:Number):void {
        if (value <= 30) {
            _maxFontSize = value;
        } else _maxFontSize = 30;

        // Create event object.
        var eventObj:Event = new Event("maxFontSizeChanged");
        dispatchEvent(eventObj);

    }
```

In this example, the setter updates the value of the property, and then creates and dispatches an event to invoke an update of the destination of the data binding.

In an MXML file, you can make all public properties that you defined as variables usable as the source for data binding by including the `[Bindable]` metadata tag in an `<fx:Metadata>` block, as the following example shows:

```
<fx:Metadata>
    [Bindable]
</fx:Metadata>
```

You can also use the `[Bindable]` metadata tag in an `<fx:Script>` block in an MXML file to make individual properties that you defined as variables usable as the source for a data binding expression. Alternatively, you can use the `[Bindable]` metadata tag with properties that you defined by using setter and getter methods.

Using read-only properties as the source for data binding

You can automatically use a read-only property defined by a getter method, which means no setter method, as the source for a data-binding expression. Flex performs the data binding once when the application starts.

Because the data binding from a read-only property occurs only once at application start up, you omit the `[Bindable]` metadata tag for the read-only property.

Using static properties as the source for data binding

You can use a static variable as the source for a data-binding expression. Flex performs the data binding once when the application starts, and again when the property changes.

You can automatically use a static constant as the source for a data-binding expression. Flex performs the data binding once when the application starts. Because the data binding occurs only once at application start up, you omit the `[Bindable]` metadata tag for the static constant. The following example uses a static constant as the source for a data-binding expression:

```
<?xml version="1.0"?>
<!-- binding/StaticBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
      <![CDATA[
        // This syntax casues a compiler error.
        // [Bindable]
        // public static var varString:String="A static var.";
        public static const constString:String="A static const.";
      ]]>
    </fx:Script>
    <!-- This binding occurs once at application startup. -->
    <s:Button label="{constString}"/>
</s:Application>
```

Working with bindable property chains

When you specify a property as the source of a data binding expression, Flex monitors not only that property for changes, but also the chain of properties leading up to it. The entire chain of properties, including the source property, is called a *bindable property chain*. In the following example, `firstName.text` is a bindable property chain that includes both a `firstName` object and its `text` property:

```
<s:Label id="myText" text="{firstName.text}"/>
```

You can have a fairly long bindable property chain, as the following example shows:

```
<s:Label id="myText" text="{user.name.firstName.text}"/>
```

For the data binding mechanism to detect changes to the `text` property, only the `text` property has to be bindable. However, if you want to assign a new value to any part of the chain at runtime, every element in the chain must be bindable. Otherwise, modifying the `user`, `name`, or `firstName` property at runtime results in the data binding mechanism no longer being able to detect changes to the `text` property.

When using the `BindingUtils.bindProperty()` or `BindingUtils.bindSetter()` method, you specify the bindable property chain as an argument to the method. For example, the `bindProperty()` method has the following signature:

```
public static function bindProperty(site:Object, prop:String, host:Object, chain:Object,
    commitOnly:Boolean = false):ChangeWatcher
```

The *`host`* and *`chain`* arguments specify the source of the data binding expression. You can define a data binding expression by using the `bindProperty()` method, as the following example shows:

```
bindProperty(myText, 'text', user, ["name","firstName","text"]);
```

In this example, `["name","firstName","text"]` defines the bindable property chain relative to the user object. Notice that `user` is not part of the bindable property change in this example.

In an MXML data-binding expression, the bindable property chain is always relative to `this`. Therefore, to define a data binding equivalent to the MXML data binding expression shown above, you write the `bindProperty()` method as the following example shows:

```
bindProperty(myText, 'text', this, ["user", "name","firstName","text"]);
```

Considerations for using the binding feature

Consider the following when using the binding feature:

- You cannot bind to style properties, but you can bind to the `getStyle()` method, as the following example shows:

  ```
  <s:Button label="Button 1" id="b1" color="green"/>
  <s:Button label="Button 2" color="{b1.getStyle('color')}"/>
  <s:Button label="Button 3" click="b1.setStyle('color', 'red');"/>
  ```

  In this example, Button 2 uses data binding to set its `color` style property to the same value as the `color` style of Button 1.

- If you bind a model into the `dataProvider` property of a component, you should not change items in the model directly. Instead, change the items through the Collections API. Otherwise, the component to which the model is bound is not redrawn to show the changes to the model. For example, instead of using the following:

  ```
  myGrid.getItemAt(itemIndex).myField = 1;
  ```

  You would use the following:

  ```
  myGrid.dataProvider.editField(itemIndex, "myField", 1);
  ```

- Array elements cannot function as binding sources at run time. Arrays that are bound do not stay updated if individual fields of a source Array change. Binding copies values during instantiation after variables are declared in an `<fx:Script>` tag, but before event listeners execute.

Debugging data binding

In some situations, data binding may not appear to function correctly, and you may need to debug them. The following list contains suggestions for resolving data binding issues:

1 Pay attention to warnings.

  It is easy to see a warning and think that it doesn't matter, especially if binding appears to work at startup, but warnings are important.

  If a warning is about a missing `[Bindable]` on a getter/setter property, even if the binding works at startup, subsequent changes to the property are not noticed.

  If a warning is about a static variable or a built-in property, changes aren't noticed.

2 Ensure that the source of the binding actually changed.

  When your source is part of a larger procedure, it is easy to miss the fact that you never assigned the source.

3 Ensure that the bindable event is being dispatched.

  You can use the Flex command-line debugger (fdb), or the Adobe® Flash® Builder™ debugger to make sure that the `dispatchEvent()` method is called. Also, you can add a normal event listener to that class to make sure it gets called. If you want to add the event listener as a tag attribute, you must place the `[Event('`*myEvent*`')]` metadata at the top of your class definition or in an `<fx:Metadata>` tag in your MXML.

4 Create a setter function and use a `<fx:Binding>` tag to assign into it.

  You can then put a trace or an alert or some other debugging code in the setter with the value that is being assigned. This technique ensures that the binding itself is working. If the setter is called with the right information, you will know that it's your destination that is failing, and you can start debugging there.

Introduction to containers

Containers provide a hierarchical structure that lets you control the layout characteristics of child components.

About containers

A *container* defines a rectangular region of the drawing surface of Adobe® Flash® Player. Within this region, you define the components, including controls and containers, that you want to appear. Components defined within a container are called *children*. Adobe® Flex® provides a wide variety of containers, ranging from simple boxes, panels, and forms to accordions that provide built-in navigation among its children.

At the root of an application written in Flex is a single container, called the application container, that represents the entire Flash Player drawing surface. The application container holds all other containers and controls. For more information, see "Application containers" on page 393.

A container uses a set of rules to control the layout of its children, including sizing and positioning. Flex defines layout rules to simplify the design and implementation of rich Internet applications, while also providing enough flexibility to let you create a diverse set of applications.

About layout containers and navigator containers

Flex defines two types of containers:

Layout containers Control the sizing and positioning of the children defined within them. Children can be controls or other containers. For more information on Spark layout containers, see "Spark containers" on page 417. For more information on MX layout containers, see "MX layout containers" on page 574.

Navigator containers Control user movement, or navigation, among multiple child containers. Navigators only take MX containers or the Spark NavigatorContent container as children. The individual child containers, not the navigator container itself, control the layout and positioning of their children. For more information, see "MX navigator containers" on page 628.

About Spark and MX containers

Flex defines two sets of containers: MX and Spark. The Spark containers are defined in the spark.components package. The MX containers are defined in the mx.core and mx.containers packages.

While you can use MX containers to perform most of the same layout that you can perform using the Spark containers, Adobe recommends that you use the Spark containers when possible.

About container children

One of the main differences between Spark and MX containers is that an MX container can only use components that implement the IUIComponent interface as its children. The UIComponent class implements the IUIComponent interface, so you can use all Flex components and containers as children of an MX container.

The Spark Group and SkinnableContainer containers can hold children that implement the IVisualElement interface. These components include all subclasses of the UIComponent class, and any subclasses of the GraphicElement class. Many of the classes in the spark.primitives package, such as Rect and Ellipse, are subclasses of the GraphicElement class. Therefore, you can use those classes as children of a Spark container to add graphics to the container.

The Spark SkinnableDataContainer and Spark DataGroup containers can hold children that implement the IUIComponent interface, and hold data items such as a String, Number, or Object. You must define an item renderer to control the display of the data item. For more information, see "The Spark DataGroup and Spark SkinnableDataContainer containers" on page 466.

About layout

Containers implement a default layout algorithm so you do not have to spend time defining it. Instead, you concentrate on the information that you deliver, and the options that you provide for your users, and not worry about implementing all the details of user action and application response. In this way, Flex provides the structure that lets you quickly and easily develop an application with a rich set of features and interactions.

Layout rules also offer the advantage that your users soon grow accustomed to them. That is, by standardizing the rules of user interaction, your users do not have to think about how to navigate the application, but can instead concentrate on the content that the application offers.

One of the main differences between Spark and MX containers is that the layout algorithm for MX containers is fixed, but for Spark containers it is selectable. Therefore, MX defines a different container for each type of layout. Spark defines a smaller set of containers, but lets you switch the layout algorithm to make them more flexible.

The MX Canvas, MX Application, and MX Panel containers, and all Spark containers, can use *absolute* layout. With absolute layout, you explicitly specify the children's *x* and *y* positions in the container. Alternatively, you can use *constraint-based layout* to anchor the sides, baseline, or center of the children relative to the parent.

Absolute layout provides a greater level of control over sizing and positioning than does automatic layout. But absolute layout provides this control at the cost of making you explicitly specify the positions of all container children.

Spark and MX containers let you define custom layout rules. For Spark, you can define a custom layout class and apply it to a Spark container. For MX, create a subclass of one of the MX containers and override its layout mechanism.

For more information on layout, see "Laying out components" on page 359.

About skinning

Skinning is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of bitmap images, SWF files, or vector images. All MX containers are skinnable.

Most, but not all, Spark containers are skinnable. The Spark Group and DataGroup containers are not skinnable. These containers provide a lightweight mechanism that you use to perform layout. They do not support skinning to ensure that they add minimal overhead to your application. If you need to modify the visual appearance, use a skinnable Spark container instead.

For more information, see "Spark Skinning" on page 1602"Improving startup performance" on page 2333 and "Skinning MX components" on page 1655.

About the creation policy

Containers define a creation policy that specifies when its children are created. By default, a container delays creating its children until they are needed by the application. This process is called *deferred instantiation*. Since all children are not created at application start up, your application appears to run faster.

You can control the creation policy for a container. For example, you can create all children at startup, or you can explicitly determine when a child is created.

You can use the following creation policies:

- `ContainerCreationPolicy.AUTO` The container delays creating children until they are needed.

  For navigator containers such as the MX ViewStack, MX TabNavigator, and MX Accordion containers, the container creates its direct children immediately, but waits to create the descendants of each child until the child needs to be displayed. As a result, only the initially required child or children of a container get processed past the preinitialization stage. An auto creation policy produces the best startup time because fewer components are created initially.

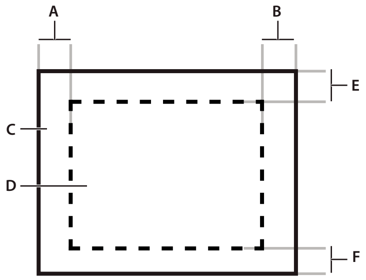
- `ContainerCreationPolicy.ALL` All container children are created and initialized before the container is initialized.
- `ContainerCreationPolicy.NONE` Requires the container to explicitly create every child instance.

For detailed information on creation policies, see "Improving startup performance" on page 2333.

Using containers

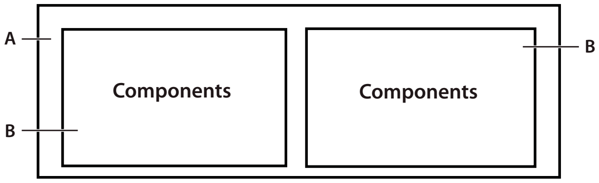
The rectangular region of a container encloses its *content area*, the area that contains its child components. The size of the region around the content area is defined by the container. That region can include extra space, or padding, around the content area and an optional border around the container.

The following image shows a container and its content area, padding, and borders:



A. *Left padding* ***B.*** *Right padding* ***C.*** *Container* ***D.*** *Content area* ***E.*** *Top padding* ***F.*** *Bottom padding*

Although you can create an entire application by using a single container, typical applications use multiple containers. For example, the following image shows an application that uses three containers:

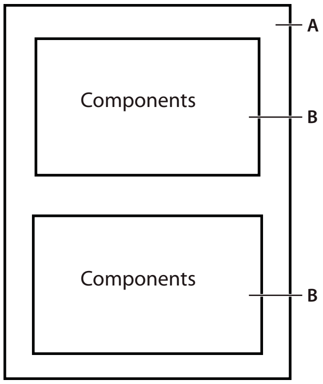


A. *Parent container* ***B.*** *Child containers*

In this example, the two child containers are nested within a parent container and are referred to as children of the parent container. The child containers themselves contain child UI components.

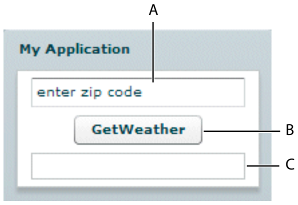
The parent container in the previous figure arranges its children in a single horizontal row and oversees the sizing and positioning characteristics of the child containers. For example, you can control the distance, or gap, between children in a container by using the `horizontalGap` and `verticalGap` properties. The two child containers are responsible for laying out their own children.

The following image shows the preceding example with the parent container configured to use a vertical layout:



A. *Parent container* *B.* *Child containers*

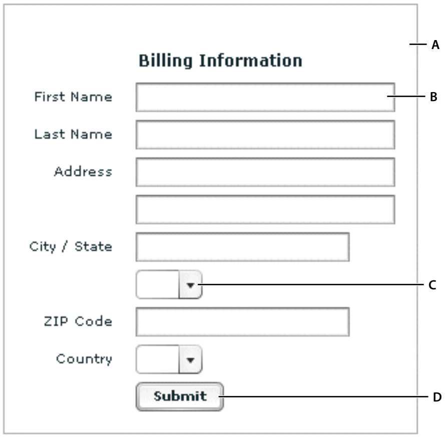
The primary use of a container is to arrange its children, where the children are either controls or other containers. The following image shows a Spark Panel container that has three child components:



A. *TextInput control* *B.* *Button control* *C.* *TextArea control*

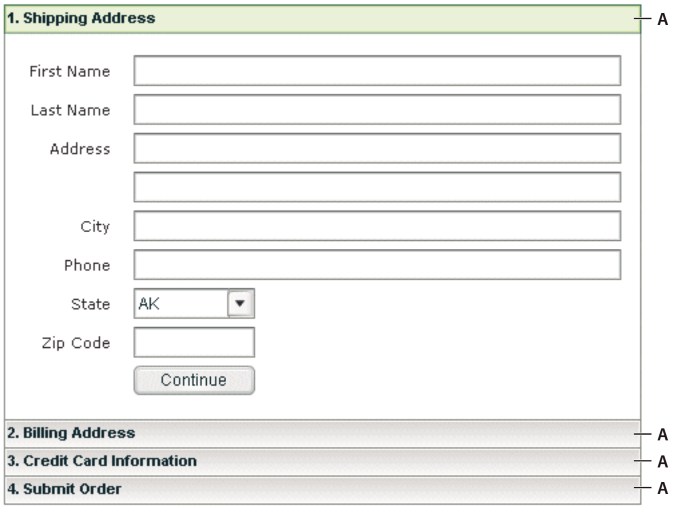
In this example, a user enters a ZIP code into the TextInput control, and then clicks the Button control to see the current temperature for the specified ZIP code in the TextArea control.

Flex supports form-based applications through its MX Form layout container. In a Form container, Flex can automatically align labels, uniformly size text input controls, and display input error notifications. The following image shows a Form container:



A. *Form container* *B.* *TextInput control* *C.* *ComboBox control* *D.* *Button control*

Navigator containers, such as the MX TabNavigator and MX Accordion containers, have built-in navigation controls that let you organize information from multiple child containers in a way that makes it easy for a user to move through it. The following image shows an Accordion container:



***A.** Accordion buttons*

You use the Accordion buttons to move among the different child containers. The Accordion container defines a sequence of child panels, but displays only one panel at a time. To navigate a container, the user clicks the navigation button that corresponds to the child panel that they want to access.

Accordion containers support the creation of multistep procedures. The preceding image shows an Accordion container that defines four panels of a complex form. To complete the form, the user enters data into all four panels. Accordion containers let users enter information in the first panel, click the Accordion button to move to the second panel, and then move back to the first if they want to edit the information. For more information, see "MX Accordion navigator container" on page 639.

Spark containers

The following table describes the Spark containers defined in the spark.components package:

Container	Type	Description	For more information
Application	Layout	The first container in an application. The default layout is absolute.	"Application containers" on page 393
BorderContainer	Layout	Defines a set of CSS styles that control the appearance of the border and background fill of the container.	"The Spark BorderContainer container" on page 430
DataGroup	Layout	Simple container that lays out it children, including data items, based on the specified layout. The default layout is absolute.	"The Spark DataGroup and Spark SkinnableDataContainer containers" on page 466
Group	Layout	Simple container that lays out it children, including graphic children, based on the specified layout. The default layout is absolute. Flex defines subclasses of Group: HGroup with horizontal layout, VGroup with vertical layout, and TileGroup with tile layout.	"The Spark Group and Spark SkinnableContainer containers" on page 425
NavigatorContent	Layout	Simple container that can be used in an MX navigator container, such as the ViewStack, TabNavigator and Accordion containers.	"The Spark NavigatorContent container" on page 433
Panel	Layout	Displays a title bar, a caption, a border, and its children. The default layout is absolute.	"The Spark Panel container" on page 434

Container	Type	Description	For more information
SkinnableContainer	Layout	Skinnable container that lays out it children, including graphic children, based on the specified layout. This container supports skinning. The default layout is vertical.	"The Spark Group and Spark SkinnableContainer containers" on page 425
SkinnableDataContainer	Layout	Skinnable container that lays out it children, including data items, based on the specified layout. This container supports skinning. The default layout is vertical.	"The Spark DataGroup and Spark SkinnableDataContainer containers" on page 466
SkinnablePopUpContainer	Layout	Skinnable container that lays out its children in a skinnable container opened as a pop up window.	"The Spark SkinnablePopUpContainer container" on page 437
TitleWindow	Layout	Skinnable Panel container that is optimized for use as a pop-up window.	"The Spark TitleWindow container" on page 443

MX containers

The following table describes the MX containers defined in the mx.core and mx.components packages:

Container	Type	Description	For more information
Accordion	Navigator	Organizes information in a series of child panels, where one panel is active at any time.	"MX Accordion navigator container" on page 639
Application	Layout	The first container in an application. The default layout is absolute.	"Application containers" on page 393
ApplicationControlBar	Layout	Holds components that provide global navigation and application commands. Can be docked at the top of an Application container.	"MX ApplicationControlBar layout container" on page 582
Box (HBox and VBox)	Layout	Displays content in a uniformly spaced row or column. An HBox container horizontally aligns its children; a VBox container vertically aligns its children. Adobe recommends that you use the Spark containers when possible, instead of the MX Box container.	"MX Box, HBox, and VBox layout containers" on page 579
Canvas	Layout	Defines a container in which you must explicitly position its children. Adobe recommends that you use the Spark containers with BasicLayout when possible, instead of the MX Canvas container.	"MX Canvas layout container" on page 574
ControlBar	Layout	Places controls at the lower edge of a Panel or TitleWindow container.	"MX ControlBar layout container" on page 581
DividedBox (HDividedBox and VDividedBox)	Layout	Lays out its children horizontally or vertically, much like a Box container, except that it inserts an adjustable divider between the children.	"MX DividedBox, HDividedBox, and VDividedBox layout containers" on page 584
Form	Layout	Arranges its children in a standard form format.	"MX Form, FormHeading, and FormItem layout containers" on page 586
Grid	Layout	Arranges children as rows and columns of cells, much like an HTML table.	"MX Grid layout container" on page 603

Container	Type	Description	For more information
Panel	Layout	Displays a title bar, a caption, a border, and its children. Adobe recommends that you use the Spark Panel container when possible, instead of the MX Panel container.	"MX Panel layout container" on page 608
TabNavigator	Navigator	Displays a container with tabs to let users switch between different content areas.	"MX TabNavigator container" on page 636
Tile	Layout	Defines a layout that arranges its children in multiple rows or columns. Adobe recommends that you use the Spark containers with the TileLayout when possible, instead of the MX Tile container.	"MX Tile layout container" on page 612
TitleWindow	Layout	Displays a popup window that contains a title bar, a caption, border, a close button, and its children. The user can move the container.	"MX TitleWindow layout container" on page 614
ViewStack	Navigator	Defines a stack of containers that displays a single container at a time.	"MX ViewStack navigator container" on page 630

Container example

The following example creates an application that uses a Spark Group container with three child controls, where the Group container lays out its children vertically:

```
<?xml version="1.0"?>
<!-- containers\intro\Panel3Children.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:TextInput id="myinput" text="enter zip code"/>
        <s:Button id="mybutton" label="GetWeather"/>
        <s:TextArea id="mytext" height="20"/>
    </s:Group>
</s:Application>
```

The Group container uses basic layout by default. Use the `layout` property of a Spark container to configure its layout. In the previous example, you set the layout of the Group container to vertical.

The Group container has subclasses that you can use as shortcuts in your application: HGroup defines a Group container with horizontal layout, VGroup defines a Group container with veritical layout, and TileGroup defines a Group container with tile layout. The following example modifies the previous example to use the VGroup container:

```
<?xml version="1.0"?>
<!-- containers\intro\Panel3Children.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:VGroup>
        <s:TextInput id="myinput" text="enter zip code"/>
        <s:Button id="mybutton" label="GetWeather"/>
        <s:TextArea id="mytext" height="20"/>
    </s:VGroup>
</s:Application>
```

To actually retrieve weather information, you must set up a web service, pass it the entered ZIP code from a `click` event, and use the returned information to populate the TextArea control.

Using container events

All containers and components support events. Containers dispatch the following events:

Spark container	MX container	Description
`elementAdd`	`childAdd`	Dispatched after a child is added to the container.
`elementRemove`	`childRemove`	Dispatched before a child is removed from the container.
	`childIndexChanged`	Dispatched after a child's index in the container has changed. Spark containers dispatch multiple `itemAdd` and `itemRemove` events.
	`scroll`	Dispatched when the user manually scrolls the container.
`contentCreationComplete`		Dispatched after all container children are created, and before the children dispatch the `creationComplete` event. This event is not available for Group and DataGroup.

All components dispatch the following events after they are added to or removed from a container:

Event	Description
`add`	Dispatched by a component after the component has been added to its container and the parent and the child are in a consistent state. This event is dispatched after the container has dispatched the `childAdd` event and all changes that need to be made as a result of the addition have happened.
`remove`	Dispatched by a component after the component has been removed from its parent container. This event is dispatched after the container has dispatched the `childRemove` event and all changes that need to be made as a result of the removal have happened.

Several events are dispatched by all components, but need special consideration when dispatched by containers. This is particularly true for navigator containers, such as TabNavigator, where some children might not be created when the container is created. These events include the following:

Event	Description
`preinitialize`	Dispatched when the component has been attached to its parent container, but before the component has been initialized, or any of its children have been created. In most cases, this event is dispatched too early for an application to use to configure a component.
`initialize`	Dispatched when a component has finished its construction and its initialization properties have been set. At this point, all the component's immediate children have been created (they have at least dispatched their `preinitialize` event), but they have not been laid out. Exactly when initialize events are dispatched depends on the container's creation policy. Flex dispatches the `initialize` event for a container after it attaches all the container's direct child controls and the container's initially required children have dispatched a `preinitialize` event. When a container or control dispatches the `initialize` event, its initial properties have been set, but its width and height have not yet been calculated, and its position has not been calculated. The `initialize` event is useful for configuring a container's children. For example, you can use the container's `initialize` event to programmatically add children or set a container scroll bar's styles. You can use a container or component's `initialize` event to initialize the data provider for a control.
`creationComplete`	Dispatched when the component, and all its child components, and all their children, and so on, have been created, laid out, and are visible. Flex dispatches the `creationComplete` event for a container when those children that are initially required are fully processed and drawn on the screen, including all required children of the children, and so on. Create a listener for the `creationComplete` event, for example, if you must have the children's dimensions and positions in your event handler. Do not use the `creationComplete` event for actions that set layout properties, as doing so results in excess processing time.

The following example uses the `creationComplete` event to open an Alert box when the children of the HGroup container are fully processed and drawn:

```
<?xml version="1.0"?>
<!-- containers\intro\Panel3ChildrenEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;

            private function handleCreationComplete():void {
                Alert.show("MyHGroup created.");
            }
        ]]>
    </fx:Script>
    <s:HGroup id="MyHGroup"
        creationComplete="handleCreationComplete();">
        <s:TextInput id="myinput" text="enter zip code"/>
        <s:Button id="mybutton" label="GetWeather"/>
        <s:TextArea id="mytext" height="20"/>
    </s:HGroup>
</s:Application>
```

To better understand the order in which Flex dispatches events, consider the following application outline.

```
OuterSkinnableContainer
    InnerSkinnableContainer1
        InnerLabel1
    InnerSkinnableContainer2
        InnerLabel2
```

The `preinitialize`, `initialize`, `contentCreationComplete`, and `creationComplete` events for the containers and controls are dispatched in the following order. The indentation corresponds to the indentation in the previous outline:

```
OuterSkinnableContainer preinitialize
    InnerSkinnableContainer1 preinitialize
        InnerLabel1 preinitialize
        InnerLabel1 initialize
    InnerSkinnableContainer1 contentCreationComplete
    InnerSkinnableContainer1 initialize
    InnerSkinnableContainer2 preinitialize
        InnerLabel2 preinitialize
        InnerLabel2 initialize
    InnerSkinnableContainer2 contentCreationComplete
    InnerSkinnableContainer2 initialize
OuterSkinnableContainer contentCreationComplete
OuterSkinnableContainer initialize
        InnerLabel2 creationComplete
        InnerLabel1 creationComplete
    InnerSkinnableContainer2 creationComplete
    InnerSkinnableContainer1 creationComplete
OuterSkinnableContainer creationComplete
```

Notice that for the terminal controls, such as the Label controls, the controls are preinitialized and then immediately initialized. For containers, preinitialization starts with the outermost container and works inward on the first branch, and then initialization works outward on the same branch. This process continues until all initialization is completed.

Then, the `creationComplete` event is dispatched first by the leaf components, and then by their parents, and so on, until the application dispatches the `creationComplete` event. The order of the `creationComplete` event is the most deeply nested components dispatch the event first, then the next most deeply nested, up to the application container. For components at the same nesting level, the order of the `creationComplete` events is undefined.

The `initialize` event is useful with a container that is an immediate child of a navigator container with the `ContainerCreationPolicy.AUTO` creation policy. For example, by default, when an MX ViewStack is initialized, the first visible child container dispatches an `initialize` event. Then, as the user moves to each additional child of the container, the event gets dispatched for that child container.

The following example defines an event listener for the `creationComplete` event, which is dispatched when the user first navigates to Pane 2 of an MX Accordion navigator container:

```
<?xml version="1.0"?>
<!-- containers\intro\AccordionInitEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            public function pane2_initialize():void {
                Alert.show("Pane 2 has been created.");
            }
        ]]>
    </fx:Script>
    <mx:Accordion width="200" height="100" creationPolicy="auto">
        <mx:VBox id="pane1" label="Pane 1">
            <mx:Label text="This is pane 1."/>
        </mx:VBox>
        <mx:VBox id="pane2"
            label="Pane 2"
            creationComplete="pane2_initialize();">
            <mx:Label text="This is pane 2."/>
        </mx:VBox>
    </mx:Accordion>
</s:Application>
```

Disabling containers

All containers support the `enabled` property. By default, this property is set to `true` to enable user interaction with the container and with the container's children. If you set `enabled` to `false`, Flex dims the color of the container and of all its children, and blocks user input to the container and to all its children.

Defining a default button

You use the `defaultButton` property of a container to define a default Button control within a container. Pressing the Enter key while focus is on any control activates the Button control as if it was explicitly selected.

For example, a login form displays TextInput controls for a user name and password and a submit Button control. Typically, the user types a user name, tabs to the password field, types the password, and presses the Enter key to submit the login information without explicitly selecting the Button control. To define this type of interaction, set the `defaultButton` property of the Form control to the `id` of the submit Button control, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerDefaultB.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            public function submitLogin():void {
                text1.text="You just tried to log in.";
            }
        ]]>
    </fx:Script>
    <s:Panel title="Default Button Example">
        <s:Form defaultButton="{mySubmitBtn}">
            <s:FormItem label="Username:">
                <s:TextInput id="username" width="100"/>
            </s:FormItem>
            <s:FormItem label="Password:">
                <s:TextInput id="password" width="100"
                    displayAsPassword="true"/>
            </s:FormItem>
            <s:FormItem>
                <s:Button id="mySubmitBtn" label="Login"
                    click="submitLogin();"/>
            </s:FormItem>
        </s:Form>
    </s:Panel>
    <s:Label id="text1" width="150"/>
</s:Application>
```

***Note:** The Enter key has a special purpose in the ComboBox control. When the drop-down list of a ComboBox control is open, pressing Enter selects the currently highlighted item in the ComboBox control; it does not activate the default button. Also, when the cursor is in a TextArea control, pressing Enter adds a newline; it does not activate the default button.*

Spark containers

Spark containers provide a hierarchical structure to arrange and configure their children.

Layout with Spark containers

Spark containers have a default layout scheme, but let you switch the layout to suit your application requirements. The layout classes are defined in the spark.layouts package, and include the following classes:

- BasicLayout   Use absolute positioning, or layout constraints.
- HorizontalLayout   Lay out children in a single horizontal row.
- TileLayout   Lay out children in multiple rows and columns.
- VerticalLayout   Lay out children in a single vertical column.

For example, the default layout of the Group container is BasicLayout, which means you must set the position of each container child. For the SkinnableContainer container, the default layout is BasicLayout. You can use any of the layout classes with the Spark containers.

For more information, see “Spark containers” on page 417.

Skinning Spark containers

The Spark Group, DataGroup, and BorderContainer containers do not support skinning. All other Spark containers support skinning so that you can control the visual aspects of the container.

For more information, see “Spark Skinning” on page 1602.

Scrolling Spark containers

The following example shows a Spark Group container that contains a child Image control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkNoViewportNoSB.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Group width="100" height="100">
        <s:Image width="300" height="400"
            source="@Embed(source='/assets/logo.jpg')"/>
    </s:Group>
</s:Application>
```

The size of the Image control is larger than that of its parent Group container. By default, the child extends past the boundaries of the parent container.

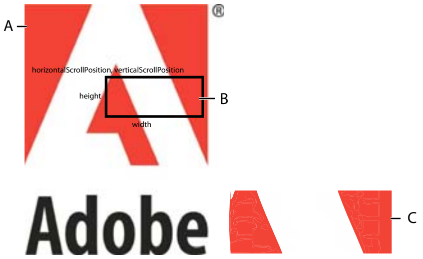
Rather than allow the child to extend past the boundaries of the parent container, you can clip the child to the container boundaries. With clipping enabled, you can control the part of the underlying image to show in the container. To clip the children to the container boundaries, use a viewport and scroll bars.

Configuring a Spark viewport

A viewport is any class that implements the IViewport interface. The Spark Group and DataGroup containers implement this interface, so they can function as viewports. All subclasses of Group and DataGroup can also function as viewports.

A viewport defines a logical content area that encompasses all of the children of the container. It also defines a rectangular area, called the visible window, of the logical content area. The visible window is the subset of the logical area of a container that you want to display.

The following figure shows a Group container with a child Image control that is larger than the area of the Group container:



A. Group container B. Area of the visible window C. Visible window of the Group container that appears in the application

The application user only sees that part of the container enclosed by the visible window.

You use the following properties of the Spark container to configure the viewport:

- `contentHeight` and `contentWidth` specify the size of the logical content area of the container. This is the area occupied by all of the container's children.
- `height` and `width` specify the size of the visible window of the container.
- `verticalScrollPosition` and `horizontalScrollPosition` specify the origin of the visible window in the container's coordinate system. The default value is (0,0) corresponding to the upper-left corner of the container.
- `clipAndEnableScrolling`, if `true`, specifies to clip the children to the boundaries of the container. If `false`, the container children extend past the container boundaries, regardless of the size specification of the container. The default value is `false`.

The following example sets the `clipAndEnableScrolling` property to `true` to clip the container's children to the container boundaries:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkViewport.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group width="100" height="100"
        clipAndEnableScrolling="true">
        <s:Image width="300" height="400"
            source="@Embed(source='/assets/logo.jpg')"/>
    </s:Group>
</s:Application>
```

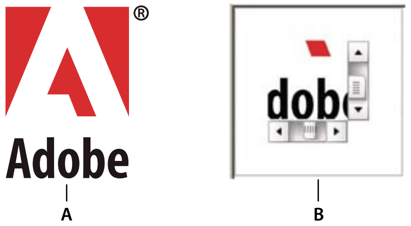
In this example, you set the `width` and `height` of the container, corresponding to the visible window, to 100 by 100 pixels.

By default, the visible window is located at the coordinates (0,0) in the container. Use the `verticalScrollPosition` and `horizontalScrollPosition` properties to set its location, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkViewportSet.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
    <s:Group width="100" height="100"
        horizontalScrollPosition="50" verticalScrollPosition="50"
        clipAndEnableScrolling="true">
        <s:Image width="300" height="400"
            source="@Embed(source='/assets/logo.jpg')"/>
    </s:Group>
</s:Application>
```

Adding scroll bars to a Spark container

The visible window of a viewport on its own is not movable by the application user. However, you can combine a viewport with scroll bars so the user can scroll the visible window to see the entire container. Scroll bars let you display an object that is larger than the available screen space or display more objects than fit in the current size of the container, as the following image shows:



A. *Image at full size* *B.* *Image in a container with scroll bars*

Add scroll bars to Spark containers in the following ways:

- For viewport containers such as Group and DataGroup, add HScrollBar and VScrollBar components.
- For viewport containers such as Group and DataGroup, wrap the container in a Scroller component. The Scroller component displays a single child component with horizontal and vertical scroll bars.
- For any skinnable Spark container, create a skin for the container that uses the Scroller component. For information on creating a scrollable skin, see "Adding scroll bars to Spark containers" on page 1637.

In the following example, you add the HScrollBar and VScrollBar components to the application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkScrollViewportSetSB.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>

    <s:VGroup>
        <s:Group id="myGroup" width="100" height="100"
            clipAndEnableScrolling="true"
            horizontalScrollPosition="50" verticalScrollPosition="50">
            <s:Image width="300" height="400"
                source="@Embed(source='/assets/logo.jpg')"/>
        </s:Group>
        <s:HScrollBar viewport="{myGroup}" width="100"/>
    </s:VGroup>
    <s:VScrollBar viewport="{myGroup}" height="100"/>
</s:Application>
```

Use the `viewport` property of a scroll bar component to specify the viewport associated with it. In this example, you associate the myGroup container with the scoll bars. As you manipulate the HScrollBar component, it modifies the `horizontalScrollPosition` property of the group container. The VScrollBar component modifies the `verticalScrollPosition` property.

In the following example, you add scroll bars to the Spark Group container by wrapping the container in the Scroller component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkScroll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >

    <s:Scroller width="100" height="100">
        <s:Group>
            <s:Image width="300" height="400"
                source="@Embed(source='/assets/logo.jpg')"/>
        </s:Group>
    </s:Scroller>
</s:Application>
```

The Scroller component can only wrap a single component. The child of a Scroller tag must implement the IViewport interface, which is implemented by the Spark Group and DataGroup containers.

The Scroller component sets the `height` and `width` properties, rather than its child. Alternatively, you can omit the size settings on the Scroller component and let the application determine its size.

To ensure that scroll bars appear, do not set explicit sizes on the child of the Scroller. If you set the size of the child explicitly, then that size becomes the size of the child and no scroll bars appear. Instead, use constraints to size the Scroller (such as setting the `height` property to 100%).

If the Scroller is unconstrained, it sizes itself to be as big as the viewport. When you constrain the size of the Scroller a size smaller than the child, the Scroller displays scroll bars so that you can view the entire child.

Notice that the Group container did not set the `clipAndEnableScrolling` property to `true`. The Scroller component automatically sets the `clipAndEnableScrolling` property to `true` for any viewport within it. Scroll bars only appear if required by the viewport.

Use the `Scroller.ensureElementIsVisible()` method to scroll a component into view. Simply pass the component as an argument to the `ensureElementIsVisible()` method.

The default skin for the scroll bars defines the minimum height of a vertical scroll bar and the minimum width of a vertical scroll bar as 35 pixels. If the size of the scroll bar is less than 35 pixels, the thumb is hidden.

The default property of the Scroller tag is `viewport`. The previous example omits this tag, but you can specify it explicitly, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkScrollViewport.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Scroller width="100" height="100">
        <s:viewport>
            <s:Group>
                <s:Image width="300" height="400"
                    source="@Embed(source='/assets/logo.jpg')"/>
            </s:Group>
        </s:viewport>
    </s:Scroller>
</s:Application>
```

You can combine scroll bars with explicit settings for the container's viewport. The viewport settings determine the initial position of the viewport, and then you can use the scroll bars to move it, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkScrollViewportSet.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
    <s:Scroller width="100" height="100">
        <s:Group
            horizontalScrollPosition="50"
            verticalScrollPosition="50">
            <s:Image width="300" height="400"
                source="@Embed(source='/assets/logo.jpg')"/>
        </s:Group>
    </s:Scroller>
</s:Application>
```

Not all Spark containers implement the IViewPort interface. Therefore, those containers, such as the BorderContainer and SkinnableContainer containers, cannot be used as the direct child of the Scroller component.

However, all Spark containers can have a Scroller component as a child component. For example, to use scroll bars on a child of the Spark BorderContainer container, wrap the child in a Scroller component, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkScrollBorder.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
    <s:BorderContainer width="100" height="100"
        borderWeight="3" borderStyle="solid">
        <s:Scroller width="100%" height="100%">
            <s:Group
                horizontalScrollPosition="50"
                verticalScrollPosition="50">
                <s:Image width="300" height="400"
                    source="@Embed(source='/assets/logo.jpg')"/>
            </s:Group>
        </s:Scroller>
    </s:BorderContainer>
</s:Application>
```

The previous example used the Scroller component to make a child of the BorderContainer container scrollable. To make the entire BorderContainer container scrollable, wrap it in a Group container. Then, make the Group container the child of the Scroller component, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkScrollBorderOuter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
    <s:Scroller width="100" height="100">
        <s:Group>
            <s:BorderContainer
                borderWeight="3" borderStyle="solid">
                    <s:Image width="300" height="400"
                        source="@Embed(source='/assets/logo.jpg')"/>
            </s:BorderContainer>
        </s:Group>
    </s:Scroller>
</s:Application>
```

MX containers

MX containers include layout containers and navigators. MX navigators can only take MX components as children.

Layout with MX containers

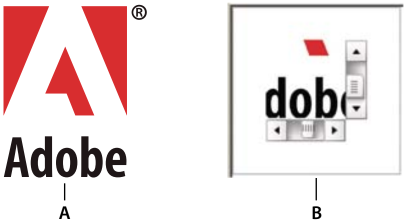
All MX containers have a predefined layout. Some containers, such as Application and Panel do let you switch among several possible layouts, but most containers do not. For example, the VBox container always lays out its children in a vertical column, and the Tile container always lays out its children multiple rows and columns.

Skinning MX containers

MX containers support skinning so that you can control the visual aspects of the container. For more information on skinning MX containers, see “Skinning MX components” on page 1655.

Scrolling MX containers

MX containers support automatic scroll bars, which let you display an object that is larger than the available screen space or display more objects than fit in the current size of the container, as the following image shows:



A. *Image at full size* ***B.*** *Image in an HBox container*

Automatic scroll bars appear by default when the container children are larger than the container, and the `clipContent` property of the container is `true`. The default value of `clipContent` is `true`.

In the following example, you use an HBox container to let users scroll an image, rather than rendering the complete image at its full size. In this example, you explicitly set the size of the HBox container to 75 by 75 pixels, a size smaller than the imported image. If you omit the sizing restrictions on the HBox container, it attempts to use its default size, which is a size large enough to hold the image:

```
<?xml version="1.0"?>
<!-- containers\intro\HBoxScroll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:HBox width="75" height="75">
        <mx:Image source="@Embed(source='assets/logo.jpg')"/>
    </mx:HBox>
</s:Application>
```

By default, Flex draws scroll bars only when the contents of a container are larger than that container. To force the container to draw scroll bars, you can set the `horizontalScrollPolicy` and `verticalScrollPolicy` properties to `on`.

Using container scroll properties

The following container properties and styles control scroll bar appearance and behavior:

- The `horizontalScrollPolicy` and `verticalScrollPolicy` properties control the display of scroll bars. By default, both properties are set to `auto`, which configures Flex to include scroll bars only when necessary. You can set these properties to `on` to configure Flex to always include scroll bars, or set the properties to `off` to configure Flex to never include scroll bars. In ActionScript, you can use constants in the ScrollPolicy class, such as `ScrollPolicy.ON`, to represent these values.
- The `horizontalLineScrollSize` and `verticalLineScrollSize` properties determine how many pixels to scroll when the user selects the scroll bar arrows. The `verticalLineScrollSize` property also controls the amount of scrolling when using the mouse wheel. The default value is five pixels.
- The `horizontalPageScrollSize` and `verticalPageScrollSize` properties determine how many pixels to scroll when the user selects the scroll bar track. The default value is 20 pixels.

  Note: *If the `clipContent` property is `false`, a container lets its child extend past its boundaries. Therefore, no scroll bars are necessary, and Flex never displays them, even if you set `horizontalScrollPolicy` and `verticalScrollPolicy` to `on`.*

Scroll bar layout considerations

Your configuration of scroll bars can affect the layout of your application. For example, if you set the horizontalScrollPolicy and verticalScrollPolicy properties to on, the container always includes scroll bars, even if they are not necessary. Each scroll bar is 16 pixels wide. Therefore, turning them on when they are not needed is similar to increasing the size of the right and bottom padding of the container by 16 pixels.

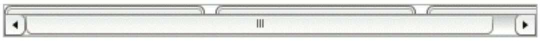
If you keep the default values of auto for the horizontalScrollPolicy and verticalScrollPolicy properties, Flex lays out the application as if the properties are set to off. That is, the scroll bars are not counted as part of the layout.

If you do not keep in mind this behavior, your application might have an inappropriate appearance. For example, if you have an HBox container that is 30 pixels high and 100 pixels wide and has two buttons that are each 22 pixels high and 40 pixels wide, the children are contained fully inside the HBox container, and no scroll bars appear. However, if you add a third button, the children exceed the width of the HBox container, and Flex adds a horizontal scroll bar at the bottom of the container. The scroll bar is 16 pixels high, which reduces the height of the content area of the container from 30 pixels to 14 pixels. This means that the Button controls, which are 22 pixels high, are too tall for the HBox, and Flex, by default, adds a vertical scroll bar.

Consider the following example:

```
<?xml version="1.0"?>
<!-- components\ScrollHBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:HBox width="400">
        <mx:Button label="Label 1"
            width="50%"
            minWidth="200"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</s:Application>
```

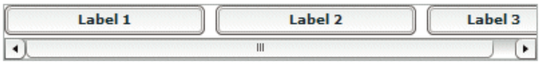
In this example, the default width of the fixed-size button is 66 pixels, so there are 324 pixels of space available for the percentage-based buttons after accounting for the gap between components. The minimum widths of the first and second buttons are greater than the percentage-based values, so Flex assigns those buttons the set widths of 200 and 150 pixels, even though the HBox container only has 324 pixels free. The HBox container uses scroll bars to provide access to its contents because they now consume more space than the container itself.



Notice that the addition of the scroll bar doesn’t increase the height of the container from its initial value. Flex considers scroll bars in its sizing calculations only if you explicitly set the scroll policy to ScrollPolicy.ON. So, if you use an auto scroll policy (the default), the scroll bar overlaps the buttons. To prevent this behavior, you can set the height property for the HBox container or allow the HBox container to resize by setting a percentage-based width. Remember that changing the height of the HBox container causes other components in your application to move and resize according to their own sizing rules. The following example adds an explicit height and permits you to see the buttons and the scroll bar:

```
<?xml version="1.0"?>
<!-- components\ScrollHBoxExplicitHeight.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:HBox width="400" height="42">
        <mx:Button label="Label 1"
            width="50%"
            minWidth="200"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</s:Application>
```

Flex draws the following application:



Alternatively, you can set the HBox control’s `horizontalScrollPolicy` property to `ScrollPolicy.ON`. This reserves space for the scroll bar during the initial layout pass, so it fits without overlapping the buttons or setting an explicit height. This also correctly handles the situation where the scroll bars change their size when you change skinning or styles. This technique places an empty scroll bar area on the container if it does not need scrolling, however.

Controlling scroll delay and interval

Scroll bars have two styles that affect how they scroll:

- The `repeatDelay` style specifies the number of milliseconds to wait after the user selects a scroll button before repeating scrolling.
- The `repeatInterval` style specifies the number of milliseconds to wait between each repeated scroll while the user keeps the scroll arrows selected.

  These settings are styles of the scroll bar subcontrol, not of the container, and, therefore, require a different treatment than properties such as `horizontalScrollPolicy`. The following example sets the scroll policy consistently for all scroll bars in the application:

```
<?xml version="1.0"?>
<!-- containers\intro\HBoxScrollDelay.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|HScrollBar, mx|VScrollBar {
            repeatDelay: 2000;
            repeatInterval:1000;
        }
    </fx:Style>
    <mx:HBox id="hb1" width="75" height="75">
        <mx:Image source="@Embed(source='assets/logo.jpg')"/>
    </mx:HBox>
</s:Application>
```

This example results in the same scrollable logo as shown in “Scrolling MX containers” on page 343, but the scroll bars behave differently. When the user clicks and holds the mouse button down over any of the scroll bar arrows or the scroll bar track, the image initially scrolls once, waits two seconds, and then scrolls at a rate of one line or page a second.

To set a style on a single scroll bar, use a line such as the following in the event listener for the `initialize` event for the application or the control with the scroll bar:

```
ScrollBar(hb1.horizontalScrollBar).setStyle("repeatDelay", 2000);
```

In this case, `hb1` is an HBox control. All containers have `horizontalScrollBar` and `verticalScrollBar` properties that represent the container’s ScrollBar subcontrols, if they exist. Cast these properties to the ScrollBar class, because their type is the IScrollBar interface, not the ScrollBar class.

Creating and managing container children at run time

You typically use MXML to lay out the user interface of your application, and use ActionScript for event handling and run-time control of the application. You can also use ActionScript to create component instances at run time. For example, you could use MXML to define an empty Accordion container and use ActionScript to add panels to the container in response to user actions.

About the display list and container children

Flash Player maintains a tree of visible (or potentially visible) objects that make up your application. The root of the tree is the Spark Application or MX Application object, and child containers and components are branches and leaf nodes of the tree. That tree is known as the *display list*. When you add child components to a container or remove child components from a container, you are adding and removing them from the display list. You can also change their relative positions by changing their positions in the display list.

Although the display list is a tree rooted at the top of the application, when you manipulate a container’s children by ’s methods and properties, you only access the container’s direct children. Treat the container’s children as items in a list, where the first child has an index of 0.

Using the container API for managing container children

Spark Group and SkinnableContainer containers, and all MX containers, provide properties and methods that you use to manage the container’s children. Because the Spark Group and SkinnableContainer containers can hold many types of children, the methods that you use to manipulate its children refer to the children by a generic name of *element*.

Note: *This section does not apply to the Spark DataGroup and Spark SkinnableDataContainer. Those containers use a data provider to define their children. You manage the children of those containers in the same way that you manage any data provider control. For more information, see "Data providers and collections" on page 898 and "The Spark DataGroup and Spark SkinnableDataContainer containers" on page 466.*

The following table shows these properties and methods:

Spark container	**MX container**	**Description**
`numElements`	`numChildren`	Number of children in the container.
`addElement()`	`addChild()`	Adds a child to the container as the last child.
`addElementAt()`	`addChildAt()`	Add a child at a specific index in the container.
	`getChildren()`	Returns an Array containing all children.
`getElementAt()`	`getChildAt()`	Return a child at the specified index.
	`getChildByName()`	Return a child with the specified id.
`getElementIndex()`	`getChildIndex()`	Returns the index of a child.
`removeAllElements()`	`removeAllChildren()`	Removes all container children.
`removeElement()`	`removeChild()`	Remove the first child.
`removeElementAt()`	`removeChildAt()`	Remove the child at the specified index
`setElementIndex()`	`setChildIndex()`	Set the index of a child.
`swapElements()`	`swapChildren()`	Swap the indexes of two children
`swapElementsAt()`	`swapChildrenAt()`	Swap the indexes of two children.

Note: *Make sure that you use the correct method for your type of container, either Spark or MX. Using the wrong method can cause unexpected results.*

Obtaining the number of child components in a container or application

Use the `numElements` or `numChildren` property to obtain a count of the number of direct child components that the container has in the display list. The following application gets the number of children in the application and a Group container. The Group control has five label controls, and therefore has five children. The Application container has the Group and Button controls as its children, and therefore has two children.

```
<?xml version="1.0"?>
<!-- containers\intro\VBoxNumChildren.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
      <![CDATA[
            // Import the Alert class.
            import mx.controls.Alert;

            public function calculateChildren():void {
                var myText:String = new String();
                myText="The Group container has " +
                    myGroup.numElements + " children.";
                myText+="\nThe application has " +
                    numElements + " children.";
                Alert.show(myText);
            }
        ]]>
    </fx:Script>
    <s:VGroup id="myGroup">
        <s:Label text="This is label 1."/>
        <s:Label text="This is label 2."/>
        <s:Label text="This is label 3."/>
        <s:Label text="This is label 4."/>
        <s:Label text="This is label 5."/>
    </s:VGroup>

    <s:Button label="Show Children" click="calculateChildren();"/>
</s:Application>
```

In the main MXML application file, the file that contains the `<s:Application>` tag, the current scope is always the Application object. Therefore, the reference to the `numElements` property without an object prefix refers to the `numElements` property of the Application object. For more information on accessing the root application, see “About scope” on page 42.

Accessing display-only children

The MX Container class defines the `rawChildren` property that contains the full display list of all the children of a container. This list includes all the container’s children, plus the DisplayObjects that implement the container’s *chrome* (display elements), such as its border and the background image.

The `numChildren` property and accessor methods let you count and access only child components. However, the container might contain style elements and skins, such as the border and background. The container’s `rawChildren` property lets you access all children of a container, including the component “content children” and the skin and style “display children.” The object returned by the `rawChildren` property implements the IChildList interface. You then use methods and properties of this interface, such as `getChildAt()`, to access and manipulate all the container’s children.

Creating and removing components at run time

To create a component instance at run time, you define it, set any properties, and then add it as a child of a parent container by calling the `addElement()` or `addChild()` method on the parent container.

The `addElement()` method has the following signature:

```
addElement(element:IVisualElement):IVisualElement
```

The `element` argument specifies the component to add to the container.

The `addChild()` method has the following signature:

```
addChild(child:DisplayObject):DisplayObject
```

The `child` argument specifies the component to add to the container.

***Note:** Although the `child` argument of the `addChild()` method is specified as type DisplayObject, the argument must implement the IUIComponent interface to be added as a child of a container. All Flex components implement this interface.*

For example, the following application creates a Group container with an Button control called myButton:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerAddChild.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import spark.components.Button;
            public function addButton():void {
                var myButton:Button = new Button();
                myButton.label = "New Button";
                myGroup.addElement(myButton);
            }
        ]]>
    </fx:Script>
    <s:Group id="myGroup" initialize="addButton();"/>
</s:Application>
```

This example creates the control when the application is loaded rather than in response to any user action. However, you could add a new button when the user presses an existing button, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerAddChild2.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import spark.components.Button;
            public function addButton():void {
                var myButton:Button = new Button();
                myButton.label = "New Button";
                myGroup.addElement(myButton);
            }
        ]]>
    </fx:Script>
    <s:HGroup id="myGroup">
        <s:Button label="Add Button" click="addButton();"/>
    </s:HGroup>
</s:Application>
```

You use the `removeElement()` or `removeChild()` method to remove a control from a container. If the child is no longer referenced anywhere else in your application after the call, it gets destroyed by a garbage collection process. The following example removes a button from the application when the user presses it:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerRemoveChild.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function removeButton():void {
                myGroup.removeElement(myButton);
            }

            private function resetApp():void {
                if (myGroup.numElements == 0) {
                    myGroup.addElement(myButton);
                }
            }
        ]]>
    </fx:Script>
    <s:Group id="myGroup">
        <s:Button id="myButton"
            label="Remove Me"
            click="removeButton();"/>
    </s:Group>
    <s:Button label="Reset" click="resetApp();"/>
</s:Application>
```

For additional methods that you can use with container children, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Example: Creating and removing a child of a container

The following example uses MXML to define a container that contains two Button controls. You use one Button control to add an CheckBox control to the container, and one Button control to delete it.

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerComponentsExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
      <![CDATA[
        // Import the CheckBox class.
        import spark.components.CheckBox;
        // Define a variable to hold the new CheckBox control.
        private var myCheckBox:CheckBox;

        // Define a variable to track if the CheckBox control
        // is in the display list.
        private var checkBoxDisplayed:Boolean = false;

        public function addCB():void {
            // Make sure the check box isn't being displayed.
            if(checkBoxDisplayed==false){
                // Create the check box if it does not exist.
                if (!myCheckBox) {
                    myCheckBox = new CheckBox();
                    myCheckBox.label = "New CheckBox";
                }
                // Add the check box.
                myGroup.addElement(myCheckBox);
                checkBoxDisplayed=true;
            }
        }

        public function delCB():void {
            // Make sure a CheckBox control exists.
            if(checkBoxDisplayed){
                myGroup.removeElement(myCheckBox);
                checkBoxDisplayed=false;
            }
        }
      ]]>
    </fx:Script>
    <s:VGroup id="myGroup">
        <s:Button label="Add CheckBox"
            click="addCB();"/>
        <s:Button label="Remove CheckBox"
            click="delCB();"/>
    </s:VGroup>
</s:Application>
```

Example: Creating and removing children of an Accordion container

The following example adds panels to and removes them from an MX Accordion container. The Accordion container initially contains one panel. Each time you select the Add HBox button, it adds a new HBox container to the Accordion container.

***Note:** MX navigators can only take MX containers and the Spark NavigatorContent container as children. You cannot use other Spark containers as direct children of a navigator container. Instead, wrap the Spark container in an MX container on in a Spark NavigatorContent container when adding the Spark container to a navigator.*

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerComponentsExample2.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            /* Import HBox class. */
            import mx.containers.HBox;

            /* Array of created containers. */
            private var myHBoxes:Array = [];
            public function addHBox():void {
                /* Create new HBox container. */
                var newHBox:HBox = new HBox();
                newHBox.label="Label: " + String(myHBoxes.length);
                /* Add it to the Accordion container, and to the
                   Array of HBox containers. */
                myHBoxes.push(myAcc.addChild(newHBox));
            }
            public function delHBox():void {
                /* If there is at least one HBox container in the Array,
                   remove it. */
                if (myHBoxes.length>= 1) {
                    myAcc.removeChild(myHBoxes.pop());
                }
            }
      ]]>
    </fx:Script>
    <s:VGroup>
        <mx:Accordion id="myAcc" height="150" width="150">
            <mx:HBox/>
        </mx:Accordion>

        <s:Button label="Add HBox" click="addHBox();"/>
        <s:Button label="Remove HBox" click="delHBox();"/>
    </s:VGroup>
</s:Application>
```

Controlling child order

You can control the order of children by adding them in a specific order. You can also control them as follows:

- By using the `addElementAt()` or `addChildAt()` method to specify where among the component's children to add a child

  ***Note:** As with the `addChild()` method, although the `child` argument of the `addChildAt()` method is specified as type DisplayObject, the argument must implement the IUIComponent interface to be added as a child of a container. All Flex components implement this interface.*

- By using the `setElementAt()` or `setChildIndex()` method to specify the location of a specific child among a component's children in the display list

The following example modifies the previous example. It uses the `addElementAt()` method to add the CheckBox control as the first child (index 0) of the VGroup. It also has a Reorder children button that uses the `setChildIndex()` method to move a CheckBox control down the display list until it is the last child in the VGroup container.

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerComponentsReorder.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
      <![CDATA[
        // Import the CheckBox and Alert classes.
        import spark.components.CheckBox;
        import mx.controls.Alert;
        // Define a variable to hold the new CheckBox control.
        private var myCheckBox:CheckBox;

        // Define a variable to track if the CheckBox control
        // is in the display list.
        private var checkBoxDisplayed:Boolean = false;

        public function addCB():void {
            // Make sure the check box isn't being displayed.
            if(checkBoxDisplayed==false){
                // Create the check box if it does not exist.
                if (!myCheckBox) {
                    myCheckBox = new CheckBox();
                    myCheckBox.label = "New CheckBox";
                }
                // Add the check box as the first child of the container.
                myGroup.addElementAt(myCheckBox, 0);
                checkBoxDisplayed=true;
            }
        }

        public function delCB():void {
            // Make sure a CheckBox control exists.
            if(checkBoxDisplayed){
                myGroup.removeElement(myCheckBox);
                checkBoxDisplayed=false;
            }
        }

        public function reorder():void {
        // Make sure a CheckBox control exists.
            if(checkBoxDisplayed==true)
            {
                // Don't try to move the check box past the end
                // of the children. Because indexes are 0 based,
                // the last child index is one less
                // than the number of children.
                if (myGroup.getElementIndex(myCheckBox) < myGroup.numElements-1)
                {
                    // Increment the checkBoxIndex variable and use it to
                    // set the index of the check box among the VBox children.
                    myGroup.setElementIndex(myCheckBox,
                        myGroup.getElementIndex(myCheckBox) + 1);
                }
            }
            else {
                Alert.show("Add the check box before you can move it");
            }
        }
      ]]>
    </fx:Script>
    <s:VGroup id="myGroup">
        <s:Button label="Add CheckBox" click="addCB();"/>
        <s:Button label="Remove CheckBox" click="delCB();"/>
        <s:Button label="Reorder children" click="reorder();"/>
    </s:VGroup>
</s:Application>
```

Flex coordinate systems

In your application, you might have to determine the location of a component, or modify its location at run time. Adobe Flash and Flex support three coordinate systems for different purposes:

- global
- local
- content

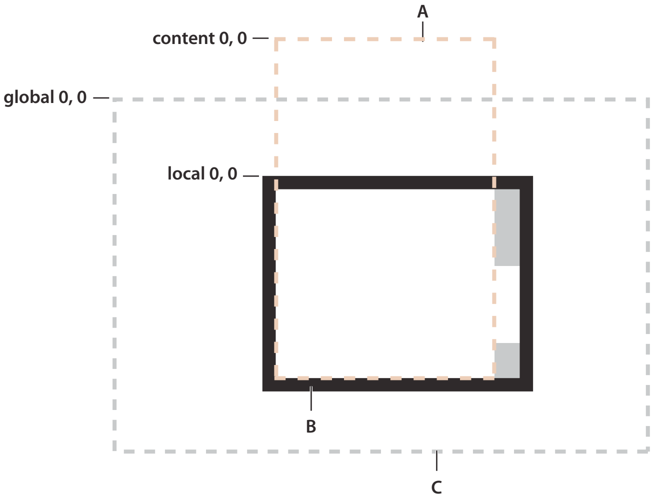
You can use Flex properties and methods to convert between coordinate systems.

About the coordinate systems

The following table describes the coordinate systems:

Coordinate system	Description
global	Coordinates are relative to the upper-left corner of the Stage in Adobe Flash Player and Adobe® AIR™, that is, the outermost edge of the application. The global coordinate system provides a universal set of coordinates that are independent of the component context. Uses for this coordinate system include determining distances between objects and as an intermediate point in converting between coordinates relative to a subcontrol into coordinates relative to a parent control. The MouseEvent class includes `stageX` and `stageY` properties that are in the global coordinate system.
local	Coordinates are relative to the upper-left corner of the component. Flex uses the local coordinate system for mouse pointer locations; all components have `mouseX` and `mouseY` properties that use the local coordinate system. The MouseEvent class includes localX and localY properties that are in the local coordinate system. Also, the Drag Manager uses local coordinates in drag-and-drop operations. The `doDrag()` method's `xOffset` and `yOffset` properties, for example, are offsets relative to the local coordinates.
content	Coordinates are relative to the upper-left corner of the component's content. Unlike the local and global coordinates, the content coordinates include all the component's content area, including any regions that are currently clipped and must be accessed by scrolling the component. Thus, if you scrolled down a Canvas container by 100 pixels, the upper-left corner of the visible content is at position 0, 100 in the content coordinates. You use the content coordinate system to set and get the positions of children of a container that uses absolute positioning. (For more information on absolute positioning, see "About component positioning" on page 361.) The UIComponent `contentMouseX` and `contentMouseY` properties report the mouse pointer location in the content coordinate system.

The following image shows these coordinate systems and how they relate to each other.



***A.** Content bounds* ***B.** Component bounds* ***C.** Stage bounds*

Using coordinate properties and methods

In some cases, you have to convert positions between coordinate systems. Examples where you convert between coordinates include the following:

- The MouseEvent class has properties that provide the mouse position in the global coordinate system and the local coordinates of the event target. You use the content coordinates to specify the locations in any container that uses absolute positioning, such as an MX Canvas container or a Spark container that uses the BasicLayout class. To determine the location of the mouse event within the Canvas container contents, not just the visible region, determine the position in the content coordinate system.

- Custom drag-and-drop handlers might have to convert between the local coordinate system and the content coordinate system when determining an object-specific drag action; for example, if you have a control with scroll bars and you want to know the drag (mouse) location over the component contents. The example in "Example: Using the mouse position in a Canvas container" on page 357 shows this use.
- Custom layout containers, where you include both visual elements, such as scroll bars or dividers, and content elements. For example, if you have a custom container that draws lines between its children, you have to know where each child is in the container's content coordinates to draw the lines.

Often, you use mouse coordinates in event handlers; when you do, you should keep in mind the following considerations:

- When you handle mouse events, it is best to use the coordinates from the MouseEvent object whenever possible, because they represent the mouse coordinates at the time the event was generated. Although you can use the container's `contentMouseX` and `contentMouseY` properties to get the mouse pointer locations in the content coordinate system, you should, instead, get the local coordinate values from the event object and convert them to the content coordinate system.
- When you use local coordinates that are reported in an event object, such as the MouseEvent `localX` and `localY` properties, remember that the event properties report the local coordinates of the mouse relative to the event target. The target component can be a subcomponent of the component in which you determine the position, such as a UITextField inside a Button component, not the component itself. In such cases, convert the local coordinates into the global coordinate system first, and then convert the global coordinates into the content coordinates container.

All Flex components provide read-only properties and methods that enable you to use and convert between coordinate systems. The following table describes these properties and methods:

Property or method	**Description**
`contentMouseX`	Returns the *x* position of the mouse, in the content coordinates of the component.
`contentMouseY`	Returns the *y* position of the mouse, in the content coordinates of the component.
`contentToGlobal (point:Point):Point`	Converts a Point object with *x* and *y* coordinates from the content coordinate system to the global coordinate system.
`contentToLocal (point:Point):Point`	Converts a Point object from the content coordinate system to the local coordinate system of the component.
`globalToContent (point:Point):Point`	Converts a Point object from the global coordinate system to the content coordinate system of the component.
`globalToLocal (point:Point):Point`	Converts a Point object from the global coordinate system to the local coordinate system of the component.
`localToContent (point:Point):Point`	Converts a Point object from the local coordinate system to the content coordinate system of the component.
`localToGlobal (point:Point):Point`	Converts a Point object from the local coordinate system to the global coordinate system.

Example: Using the mouse position in a Canvas container

The following example shows the use of the `localToGlobal()` and `globalToContent()` methods to determine the location of a mouse pointer within a Canvas container that contains multiple child Canvas containers.

This example is artificial, in that production code would use the MouseEvent class `stageX` and `stageY` properties, which represent the mouse position in the global coordinate system. The example uses the `localX` and `localY` properties, instead, to show how you can convert between local and content coordinates, including how first converting to using the global coordinates ensures the correct coordinate frame of reference.

```
<?xml version="1.0"?>
<!-- containers\intro\MousePosition.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
          import mx.controls.Alert;
          // Handle the mouseDown event generated
          // by clicking in the application.
          private function handleMouseDown(event:MouseEvent):void {

            // Convert the mouse position to global coordinates.
            // The localX and localY properties of the mouse event contain
            // the coordinates at which the event occurred relative to the
            // event target, typically one of the
            // colored internal Canvas containers.
            // A production version of this example could use the stageX
            // and stageY properties, which use the global coordinates,
            // and avoid this step.
            // This example uses the localX and localY properties only to
            // illustrate conversion between different frames of reference.
            var pt:Point = new Point(event.localX, event.localY);
            pt = event.target.localToGlobal(pt);

            // Convert the global coordinates to the content coordinates
            // inside the outer c1 Canvas container.
            pt = c1.globalToContent(pt);

            // Figure out which quadrant was clicked.
            var whichColor:String = "border area";

            if (pt.x < 150) {
                if (pt.y < 150)
                    whichColor = "red";
                else
                    whichColor = "blue";
            }
            else {
                if (pt.y < 150)
                    whichColor = "green";
                else
                    whichColor = "magenta";
            }

            Alert.show("You clicked on the " + whichColor);
          }
        ]]>
    </fx:Script>
    <!-- Canvas container with four child Canvas containers -->
    <mx:Canvas id="c1"
        borderStyle="none"
        width="300" height="300"
        mouseDown="handleMouseDown(event);">

        <mx:Canvas
```

```
            width="150" height="150"
            x="0" y="0"
            backgroundColor="red">
            <mx:Button label="I'm in Red"/>
        </mx:Canvas>
        <mx:Canvas
            width="150" height="150"
            x="150" y="0"
            backgroundColor="green">
            <mx:Button label="I'm in Green"/>
        </mx:Canvas>
        <mx:Canvas
            width="150" height="150"
            x="0" y="150"
            backgroundColor="blue">
            <mx:Button label="I'm in Blue"/>
        </mx:Canvas>
        <mx:Canvas
            width="150" height="150"
            x="150" y="150"
            backgroundColor="magenta">
            <mx:Button label="I'm in Magenta"/>
        </mx:Canvas>
    </mx:Canvas>
</s:Application>
```

Laying out components

Adobe® Flex™ lays out components by determining their sizes and positions; it provides you with multiple options for controlling both size and position.

About sizing and positioning

Component layout has two parts: component sizing and component positioning. Flex controls the layout of components by using a set of rules. These layout rules are a combination of sizing rules for individual components, and sizing and positioning rules for containers.

Because Flex defines default layout rules, you often do not have to set the size or position of components. Instead, you can concentrate on building the logic of your application and let Flex control the layout.

While Flex has default layout rules, you can use the component's properties and methods to customize the layout. For example, all components have several properties, including `height` and `width`, for specifying the component's size in absolute or container-relative terms.

Each container also has properties and styles that you can use to configure aspects of layout. You can also use different positioning techniques for laying out components in a container. For example, some containers support absolute *x*- and *y*-coordinate–based positioning.

About layout with Spark and MX containers

While the sizing and layout rules for Spark and MX containers are almost identical, there are a few differences between the two types of containers:

- Spark containers let you specify the layout class assigned to the container, such as BasicLayout, HorizontalLayout, VerticalLayout, and TileLayout.

  MX containers have a predefined layout built in to the class. Therefore, the MX VBox container always lays out its children in a vertical column, and the MX Tile container always lays out its children in rows and columns. The only exceptions to this rule are the MX Application and MX Panel containers. These containers let you switch among absolute, vertical, and horizontal layout.

- Spark containers use properties on the layout class to control the layout of the container. For example, use the `TileLayout.verticalGap` and `TileLayout.horizontalGap` properties to set the gaps between children of a Spark container that uses the TileLayout class.

  MX containers use properties and styles of the container class. Therefore, to set the gaps for a MX container, set the `verticalGap` and `horizontalGap` styles on the container.

About the Layout Manager

The Layout Manager controls layout in Flex. The manager uses the following three-stage process to determine the size and position of each component in an application:

Stage 1 - Commitment pass Determines the property settings of the application's components. This pass allows components whose contents depend on property settings to configure themselves before Flex determines their sizes and positions.

During the commitment pass, the Layout Manager causes each component to run its `commitProperties()` method, which determines the property values.

Stage 2 - Measurement pass Calculates the default size of every component in the application. This pass starts from the most deeply nested components and works out toward the application container.

The measurement pass determines the *measured*, or default, size of each component. The default size of each container is based on the default or explicit (if specified) sizes of its children. For example, an SkinnableContainer that uses the HorizontalLayout class has a default width equal to the sum of the default or explicit widths of all its children, plus the thickness of the borders, any padding, and the gaps between the children.

During the measurement pass, the Layout Manager causes each component to call its `measure()` method to determine the component's default size. All sizing properties are calculated before Flex applies any transform to the component, such as a rotation, skew, or other transform.

Stage 3 - Layout pass Lays out your application, including moving and resizing any components. This pass starts from the outermost container and works in toward the innermost component. The layout pass determines the actual size and placement of each component. It also does any programmatic drawing, such as calls to the `lineTo()` or `drawRect()` methods.

The Layout Manager causes each component to run its `updateDisplayList()` method to lay out the component's children; for this reason, this pass is also referred to as the update pass.

When lay out occurs

Flex lays out components when your application initializes. Flex also performs a layout pass when the application or a user does something that affects the sizes or positions of visual elements, including the following:

- The application changes properties that specify sizing or positioning, such as `x`, `y`, `width`, `height`, `scaleX`, `scaleY`, `rotationX`, `rotationY`, and the transform matrix.

- A change affects the calculated width or height of a component, such as when the label text for a Button control changes, or the user resizes a component.
- A child is added or removed from a container, a child is resized, or a child is moved. For example, if your application can change the size of a component, Flex updates the layout of the container to reposition its children, based on the new size of the child.
- A property or style that requires measurement and drawing, such as the `fontFamily`, changes.

Manually forcing layout

Sometimes, you might have to programmatically cause Flex to update application layout. Situations where you must force a layout include the following:

- When printing multiple page data grids by using the PrintDataGrid class.
- Before playing an effect, if the start values have just been set on the target.
- When capturing bitmap data after making property changes.

To force a layout, call the `validateNow()` method of the component to lay out. This method causes Flex to validate and update the properties, sizes, and layout of the object and all its children, and to redraw them, if necessary. Because this method is computation-intensive, be careful to call it only when it is necessary.

About component sizing

Flex provides several ways for you to control the size of controls and containers:

Default sizing Flex automatically determines the sizes of controls and containers.

Explicit sizing You set the `height` and `width` properties of the component to absolute values.

Percentage-based sizing You specify the component size as a percentage of its parent container's size.

Constraint-based layout You control size and position by anchoring component's sides to locations in their container.

For details on controlling component sizes, see "Sizing components" on page 365.

About component positioning

Flex provides two mechanisms for positioning controls:

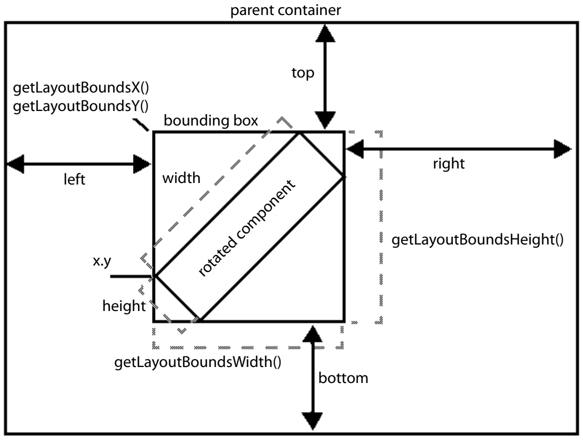
Automatic positioning Flex automatically positions a container's children according to a set of container- and component-specific rules. Most containers use automatic positioning. Automatic positioning is sometimes referred to as automatic layout. One of the differences between Spark and MX containers is that the layout algorithm for MX containers is fixed, but for Spark containers it is selectable by changing the container's `layout` property.

Absolute positioning You specify each child's `x` and `y` properties, or use a constraint-based layout that specifies the distance between one or more of the container's sides and the child's sides, baseline, or center. Absolute positioning is sometimes referred to as absolute layout. Spark containers with a layout of BasicLayout and the MX Canvas container use absolute layout. You can also choose absolute layout for the MX Application and Panel containers.

For details on controlling the positions of controls, see "Positioning components" on page 379.

About basic layout properties and methods

While Flex provides many properties that you can use to control and monitor component layout, there are a few that you use frequently. The following figure shows these properties and methods:



This figure shows how Flex lays out a component in its parent container. The `x`, `y`, `height`, and `width` properties of the component determine its size before any skins or transformations are applied to it. These are properties that you can set when defining your application.

After Flex applies any skins or transformations to the component, Flex uses the rectangular area, called the *bounding box*, to position the component in its parent container. The bounding box defines the rectangular are taken up by the component in its parent container. In the absence of any skin or transformation, the size of the bounding box is the same as the height and width of the component.

A graphical component, such as Ellipse and Rect, can define a stroke for their border. If the component defines a stroke, the center of the stroke runs along the outer edge of the component. For example, if you define a two-pixels wide border, the border appears in the area one-pixel outside the component to one-pixel inside the component. The border has to fit completely within the bounding box. Therefore, the upper-left corner of the component might be offset from the edge of the bounding box to accommodate the border.

The following table describes the basic component layout properties and methods:

Property or method	Description
`x`, `y`	The x and y coordinate of the upper-left corner of the component in the parent container. You can set these properties for the children of a container that use absolute layout. Otherwise, Flex calculates these values for you.
`height`, `width`	The height and width of the component. You can set these properties as a pixel value or as a percent of the size of the parent container. Otherwise, Flex calculates these values for you.
`top`, `bottom`, `left`, `right`	The distances between the sides of the component and sides of the parent container. Use these properties when performing constraint-based layout. Only containers that use absolute layout support constraint-based layout. For more information, see "Using constraints to control component layout" on page 384.
`getLayoutBoundsX()`, `getLayoutBoundsY()`	The x and y coordinates of the bounding box of the component in the parent container after any skins and transformations have been applied to the component.
`getlayoutBoundsHeight()`, `getLayoutBoundsWidth()`	The height hand width of the bounding box of the component in the parent container after any skins and transformations have been applied to the component. \

Basic layout rules and considerations

Flex performs layout according to the following basic rules. If you remember these rules, you should be able to easily understand the details of Flex layout. These rules help you determine why Flex lays out your application as it does and to determine how to modify your application appearance.

For a detailed description of how Flex sizes components, see “Determining and controlling component sizes” on page 367. For detailed information on component positioning, see “Positioning components” on page 379.

- Flex first determines all components’ measured (default) or explicitly set sizes *up*, from the innermost child components to the outermost (Application) component. This is done in the measurement pass.
- After the measurement pass, Flex determines all percentage-based sizes and constraints applied to the components. Flex then lays out components *down*, from the outermost container to the innermost components. This is done in the layout pass.
- Sizes that you set to a pixel value are mandatory and fixed, and override any maximum or minimum size specifications that you set for the component.
- The default sizes determined in the measurement pass specify the sizes of components that do not have explicit or percentage-based sizes (or use constraint-based layout), and are fixed.
- Percentage-based size specifications are advisory, and are relative to the size of the parent container. The layout algorithms satisfy the request if possible, and use the percentage values to determine proportional sizes, but the actual sizes can be less than the requested sizes. Percentage-based sizes are always within the component’s maximum and minimum sizes, and, subject to those bounds, don’t cause a container’s children to exceed the container size.

Component layout patterns

Flex uses different patterns to lay out different containers and their children. These patterns generally fit in the type categories listed in the following table. The table describes the general layout behavior for each type, how the default size of the container is determined, and how Flex sizes percentage-based children.

Pattern	**Container type**	**Default layout behavior**
Absolute positioning	Spark container using BasicLayout MX Canvas container or Application or Panel container with `layout="absolute"`	**General layout:** Children of the container do not interact. That is, children can overlap and the position of one child does not affect the position of any other child. You specify the child positions explicitly or use constraints to anchor the sides, baselines, or centers of the children relative to the parent container. **Default sizing:** The measurement pass finds the child with the lowest bottom edge and the child with the rightmost edge, and uses these values to determine the container size. **Percentage-based children:** Sizing uses different rules depending on whether you use constraint-based layout or *x*- and *y*- coordinate positioning. See "Flex component sizing techniques" on page 371.
Containers that arrange children linearly	Spark containers using HorizontalLayout or VerticalLayout MX Box, DividedBox, Form, Panel, TitleWindow containers	**General layout:** All children of the container are arranged in a single row or column. Each child's height and width can differ from all other children's heights or widths. **Default sizing:** The container fits the default or explicit sizes of all children and all gaps, borders, and padding. **Percentage-based children:** If children with percentage-based sizing request more than the available space, the actual sizes are set to fit in the space, proportionate to the requested percentages.

Pattern	**Container type**	**Default layout behavior**
Grid	MX Grid container	**General layout:** A container with a vertical layout of rows. Each row is a container with a horizontal layout of child controls, where all items are constrained to align with each other. The heights of all the cells in a single row are the same, but each row can have a different height. The widths of all cells in a single column are the same, but each column can have a different width. You can define a different number of cells for each row or each column of the Grid container, and individual cells can span columns or rows. **Default sizing:** The grid fits the individual rows and children at their default sizes. **Percentage-based children:** If children use percentage-based sizing, the sizing rules fit the children GridItem components within their rows, and GridRow components within the grid size according to linear container sizing rules.
Tile	Spark container using TileLayout MX Tile container	**General layout:** The container is a grid of equal-sized cells. The cells can be in row-first or column-first order. For MX, if you do not specify explicit or percentage-based dimensions, the container has as close as possible to an equal number of rows and columns, with the `direction` property determining the orientation with the larger number of items, if necessary. For Spark, if you do not specify explicit or percentage-based dimensions, the container arranges the rows and columns so that the container has an equal height and width. The `orientation` property determining the orientation with the larger number of items, if necessary. **Default sizing:** If you do not specify `tileWidth` and `tileHeight` properties (MX Tile container) or `columnWidth` and `rowHeight` (Spark Tile layout), the container uses the measured or explicit size of the largest child cell for the size of each child cell. **Percentage-based children:** The percentage-based sizes of a child component specify a percentage of the individual cell, not of the Tile container.
Navigators	MX ViewStack, Accordion, and TabNavigator containers	**General layout:** The container displays one child at a time. **Default sizing:** The container size is determined by the measured or explicit dimensions of the initially selected child, and thereafter, all children are forced to be that initial size. If you set the `resizeToChild` property to `true` the container resizes to accommodate the measured or explicit size of each child, as that child appears. **Percentage-based children:** As a general rule, you either use 100% for both height and width, which causes the children to fill the navigator bounds, or do not use percentage-based sizing.

Sizing components

Flex provides several ways for controlling the size of components. You can do the following tasks:

- Let Flex automatically determine component sizes.
- Specify pixel sizes.

- Specify component size as a percentage of the parent container.
- Combine layout and sizing by specifying a constraint-based layout.

Several Flex properties affect the size of components. As a rule, you use only a few properties for most applications, but a more complete understanding of these properties can help you understand the underlying Flex sizing mechanism and how Flex sizing properties interrelate. For information on the rules that Flex applies to determine the sizes of components based on the properties, see “Determining and controlling component sizes” on page 367.

For information on constraint-based layout, see “Using constraints to control component layout” on page 384.

Commonly used sizing properties

You typically use the following properties to specify how a component is sized:

- The `height`, `width`, `percentHeight`, and `percentWidth` properties specify the height and width of a component. In MXML tags, you use the `height` and `width` properties to specify the dimensions in pixels or as percentages of the parent container size. In ActionScript, you use the `height` and `width` properties to specify the dimensions in pixels, and use the `percentHeight` and `percentWidth` properties to specify the dimensions as a percentage of the parent container.
- The `minHeight`, `minWidth`, `maxHeight`, and `maxWidth` properties specify the minimum and maximum dimensions that a component can have if Flex determines the component size. These properties have no effect if you explicitly set the width or height in pixels.

The following characteristics and their associated properties determine the size of the component:

Characteristic	Associated properties	Description
Actual dimensions	`height`, `width`	The height and width of the displayed control, in pixels, as determined by the layout pass. If you set any explicit values, they determine the corresponding actual values.
Explicit dimensions	`explicitHeight`, `explicitWidth` Setting the `height` and `width` properties to integer values also sets the `explicitHeight` and `explicitWidth` properties.	A dimension that you set as a number of pixels. These dimensions cannot be overridden. Application developers typically use the `height` and `width` properties to set explicit dimensions. You cannot have both an explicit dimension and a percentage-based dimension; setting one unsets the other.
Percentage-based dimensions	`percentHeight`, `percentWidth` In MXML tags only, setting the `height` and `width` properties to percentage string values, such as “50%”, also sets the `percentHeight` and `percentWidth` properties.	A dimension that you set as a number in the range 0-100, as a percentage of the viewable area of the parent container. If you set a percentage-based dimension, the component is resizable, and grows or shrinks if the parent dimension changes.
Default dimensions	`measuredHeight`, `measuredWidth`	Not used directly by application developers. The dimensions of the component, as determined by the `measure()` method of the component. These values cannot be outside the range determined by the component’s maximum and minimum height and width values.

The following characteristics determine the minimum and maximum *default* and *percentage-based* dimensions that a component can have. They *do not* affect values that you set explicitly or, for MX containers only, dimensions determined by using constraint-based layout.

Characteristic	**Associated Properties**	**Description**
Minimum dimensions	`minHeight`, `minWidth` Setting the explicit minimum dimensions also sets the `minHeight` and `minWidth` properties.	The minimum dimensions a component can have. By default, Flex sets these dimensions to the values of the minimum default dimensions.
Maximum dimensions	`maxHeight`, `maxWidth` Setting the explicit maximum dimensions also sets the `maxHeight` and `maxWidth` properties.	The maximum dimensions a component can have. The default values of these properties are component-specific, but often are 10000 pixels.
Minimum default dimensions	`measuredMinHeight`, `measuredMinWidth`	Not used by application developers. The minimum valid dimensions, as determined by the `measure()` method. The default values for these properties are component-specific; for many controls, the default values are 0.

Basic sizing rules

The following rules describe how you can use Flex sizing properties to specify the size of a component:

- Any dimension property that you set overrides the corresponding default value; for example, an explicitly set `height` property overrides any default height.
- Setting the `width`, `height`, `maxWidth`, `maxHeight`, `minWidth`, or `minHeight` property to a pixel value in MXML or ActionScript also sets the corresponding explicit property, such as `explicitHeight` or `explicitMinHeight`.
- The explicit height and width and the percentage-based height and width are mutually exclusive. Setting one value sets the other to `NaN`; for example, if you set `height` or `explicitHeight` to 50 and then set `percentHeight` to 33, the value of the `explicitHeight` property is `NaN`, not 50, and the `height` property returns a value that is determined by the `percentHeight` setting.
- If you set the `height` or `width` property to a percentage value in an MXML tag, you actually set the percentage-based value, that is, the `percentHeight` or `percentWidth` property, *not* the explicit value. In ActionScript, you cannot set the `height` or `width` property to a percentage value; instead, set the `percentHeight` or `percentWidth` property.
- When you read the `height` and `width` properties at run time, the value is always the actual height or width of the control.

Determining and controlling component sizes

Flex determines the sizes of controls and containers based on the components and their properties and how you can use Flex properties to control the sizes.

Determining component size

During the measurement pass, Flex determines the components' default (also called measured) sizes. During the layout pass, Flex determines the actual sizes of the components, based on the explicit or default sizes and any percentage-based size specifications.

The following list describes sizing rules and behaviors that apply to all components, including both controls and containers. For container-specific sizing rules, see below. For detailed information on percentage-based sizing, see "Using percentage-based sizing" on page 373.

- If you specify an explicit size for any component (that is not outside the component's minimum or maximum bounds), Flex always uses that size.
- If you specify a percentage-based size for any component, Flex determines the component's actual size as part of the parent container's sizing procedure, based on the parent's size, the component's requested percentage, and the container-specific sizing and layout rules.
- The default and percentage-based sizes are always at least as large as any minimum size specifications.
- If you specify a child component size by using a percentage value and do not specify an explicit or percentage-based size for its container, the child component and the container use their default size.
- A child might require more space than is available in the parent container. The container can either let the component extend past its boundaries, or the container can clip the component to its boundaries.

  For Spark containers, the default is to let the child component extend past its boundaries. Set the `clipAndEnableScrolling` property to `true` to configure a parent to clip the child at the parent's boundaries. If you clip the children to the container boundaries, you can use the Scroller component with the container to display scroll bars. For more information, see "Scrolling Spark containers" on page 338.

  For a MX container, if a child requires more space than is available in the parent container, the parent clips the children at the parent's boundaries, and, by default, displays scroll bars on the container. Set the `clipContent` property to `false` to configure a parent to let the child extend past the parent's boundaries. Use the `scrollPolicy` property to control the display of the scroll bars. For more information, see "Scrolling MX containers" on page 343.
- If you specify a percentage-based size for a component, Flex uses the viewable area of the container in determining the sizes.
- When sizing and positioning components, Flex does not distinguish between visible and invisible components. By default, an invisible component is sized and positioned as if it were visible. To prevent Flex from considering an invisible component when it sizes and positions other components, set the component's `includeInLayout` property to `false`. This property affects the layout of the children of all containers except Accordion, FormItem, or ViewStack. For more information, see "Preventing layout of hidden controls" on page 382.

  ***Note:** Setting a component's `includeInLayout` property to `false` does not prevent Flex from displaying the component; it only prevents Flex from considering the component when it lays out other components. To prevent Flex from displaying the component, also set the visible property to `false`.*

Determining container size

Flex uses the following basic rules, in addition to the basic component sizing rules, to determine the size of a container:

- A percentage-based container size is advisory. Flex makes the container large enough to fit its children at their minimum sizes. For more information on percentage-based sizing, see "Using percentage-based sizing" on page 373.
- If you do not specify an explicit or percentage-based size for a container, Flex determines the container size by using explicit sizes that you specify for any of its children, and the default sizes for all other children.
- Flex does not consider any percentage-based settings of a container's children when sizing the container; instead, it uses the child's default size.
- If a MX container uses automatic scroll bars, Flex does not consider the size of the scroll bars when it determines the container's default size in its measurement pass. Thus, if a scroll bar is required, a default-sized container might be too small for proper appearance. For more information, see "Scrolling MX containers" on page 343.

For a Spark container, the Scroller component adds the scroll bars and are therefore separate from the container.

Each MX container and each Spark layout class for a Spark container has a set of rules that determines the container's default size. For information on default sizes of each container and layout, see the specific container sections in the *ActionScript 3.0 Reference for the Adobe Flash Platform*, and in "Application containers" on page 393; "Spark containers" on page 417; "MX layout containers" on page 574; and "MX navigator containers" on page 628.

Example: Determining a container and child sizes

The following example code shows how Flex determines the sizes of an SkinnableContainer container and its children. In this example, the SkinnableContainer uses a HorizontalLayout with three Button controls:

```
<?xml version="1.0"?>
<!-- components\HBoxSizeSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:SkinnableContainer id="hb1">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button id="b1"
            label="Label 1"/>
        <s:Button id="b2"
            label="Label 2"
            minWidth="80"/>
        <s:Button id="b3"
            label="Label 3"/>
    </s:SkinnableContainer>

    <s:Form>
        <s:FormItem label="SkinnableContainer:">
            <s:Label text="{hb1.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #1:">
            <s:Label text="{b1.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #2:">
            <s:Label text="{b2.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #3:">
            <s:Label text="{b3.width}"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

By default, the SkinnableContainer with a HorizontalLayout uses a 6 pixel gap between children. Since the Application container specifies no sizing properties, the Application container sizes itself large enough to hold the SkinnableContainer container and the Form container.

The default width for buttons is based on the label text width; in this example it is 70 pixels for all three buttons. However, the second button sets its `minWidth` property to 80 pixels. Therefore, the SkinnableContainer width is 70 + 80 + 70 + 12 = 232 pixels.

If you change the `minWidth` property of the second button to 65 pixels, the calculation uses the button's minimum width, 70 pixels, so the SkinnableContainer width is 222 pixels.

In the following example, you explicitly set the width of the Application container to 600 pixels. The Application container defines no left or right padding so the content area of its children is the entire area of the Application container. The SkinnableContainer width is now 450 pixels, 75% of 600 pixels.

***Note:** The MX Application container defines 24 pixels of right and left padding around its content area. Therefore, if the MX Application container has a width of 600 pixels, the maximum width of a child is 552 pixels. The container in this example would then be 414 pixels wide, 75% of 552 pixels.*

The button sizes are 92, 276, and 70 pixels, respectively. The third button uses the default size. The first and second buttons use 25% and 75% of the remaining available space in the container after deducting the default-width button and the gaps.

```
<?xml version="1.0"?>
<!-- components\HBoxSizePercentSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:SkinnableContainer id="hb1" width="75%">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button id="b1"
            label="Label 1"
            width="25%"/>
        <s:Button id="b2"
            label="Label 2"
            width="75%"
            minWidth="80"/>
        <s:Button id="b3"
            label="Label 3"/>
    </s:SkinnableContainer>

    <s:Form>
        <s:FormItem label="SkinnableContainer:">
            <s:Label text="{hb1.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #1:">
            <s:Label text="{b1.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #2:">
            <s:Label text="{b2.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #3:">
            <s:Label text="{b3.width}"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

If you set the SkinnableContainer `width` property to 20%, however, the SkinnableContainer width is *not* 120 pixels, 20% of the Application container width. This is because 120 pixels is too small to fit the container's children. Remember that percentage-based size specifications are advisory. The layout algorithms satisfy the request if possible, but the actual sizes can be different from the requested sizes.

Instead it is 183 pixels, the sum of 21 pixels for button 1 (the default minimum size), 80 pixels for button 2 (the specified minimum size), and 70 pixels (the specified minimum size) for button 2, 16 pixels for the gaps between buttons. The buttons are 21, 80, and 70 pixels wide, respectively.

For more information and examples showing sizing of containers and children, see "Flex component sizing techniques" on page 371. For detailed information on percentage-based sizing, see "Using percentage-based sizing" on page 373.

Flex component sizing techniques

You can use default sizing, explicit sizing, and percentage-based sizing techniques to control the size of components. For information on using constraint-based layout for component sizing, see "Using constraints to control component layout" on page 384.

Using default sizing

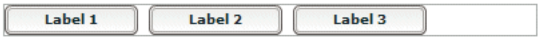
If you do not otherwise specify sizes, Flex calculates a size based on the default sizing characteristics of the particular component and the default or explicit sizes of the component's child controls.

As a rule, determine whether a component's default size (as listed for the component in the *ActionScript 3.0 Reference for the Adobe Flash Platform*) is appropriate for your application. If it is, you do not have to specify an explicit or percentage-based size.

The following example shows how you can use default sizing for Button children of an BorderContainer container. In this example, none of the children of the container specify a `width` property:

```
<?xml version="1.0"?>
<!-- components\DefaultButtonSizeSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.*">

    <s:BorderContainer width="400" height="25">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button label="Label 1"/>
        <s:Button label="Label 2"/>
        <s:Button label="Label 3"/>
    </s:BorderContainer>
</s:Application>
```

Flex, therefore, uses the default sizes of the buttons, which accommodate the button label and default padding, and draws this application as the following image shows:



Notice the empty space to the right of the third button, because the sum of the default sizes is less than the available space.

Specifying an explicit size

Use the `width` and `height` properties of a component to explicitly set its size. In this example, Flex sets the component sizes to 300 by 40 pixels.

```
<?xml version="1.0"?>
<!-- components\ExplicitTextSize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:TextInput id="myInput"
            text="This TextInput control is 300 by 40 pixels."
            width="300"
            height="40"/>
    </s:Group>
</s:Application>
```

The following example shows setting the sizes of a container and its child:

```
<?xml version="1.0"?>
<!-- components\ExplicitHBoxSize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group width="150" height="50">
        <s:TextInput id="myInput"
            text="Enter the zip code"
            width="200" height="40"/>
    </s:Group>
</s:Application>
```

Because the specified TextInput control size is larger than that of its parent container, it extends beyond the boundaries of the container. If you set the `clipAndEnableScrolling` property of the container to `true`, the child is clipped at the container boundaries.

You can also use scroll bars to see the entire TextInput control by wrapping the Group container in the Scroller control, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\ExplicitHBoxSizeScroll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Scroller>
        <s:Group width="150" height="50">
            <s:TextInput id="myInput"
                text="Enter the zip code"
                width="200" height="40"/>
        </s:Group>
    </s:Scroller>
</s:Application>
```

For more information on using scroll bars with Spark containers, see “Scrolling Spark containers” on page 338.

Note that for MX containers, the `clipContent` property of the container defaults to `true`. Therefore, MX containers clip the control at the container boundary and displays scroll bars. For more information on using scroll bars with MX containers, see “Scrolling MX containers” on page 343.

Using percentage-based sizing

Percentage-based sizing dynamically determines and maintains a component's size relative to its container; for example, you can specify that the component's width is 75% of the container. This sizing technique has several advantages over default or explicit fixed sizing:

- You only have to specify a size relative to the container; you don't have to determine exact measurements.
- The component size changes dynamically when the container size changes.
- The sizing mechanism automatically takes into account the remaining available space and fits components even if their requested size exceeds the space.

To specify a percentage value, use one of the following coding techniques:

1 In an MXML tag, set the `height` or `width` property to a percentage value; for example:

```
<s:TextArea id="ta1" width="70%" height="40%"/>
```

2 In an MXML tag or an ActionScript statement, set the `percentHeight` or `percentWidth` property to a numeric value; for example:

```
ta1.percentWidth=70;
```

The exact techniques Flex uses to determine the dimensions of a component that uses percentage-based sizing depend on the type of container. For example, a MX Tile container, or a Spark container using the TileLayout class, has cells that are all the largest default or explicit dimensions of the largest child. Child control percentage values specify a percentage of the tile cell size, not of the container size. The percentage sizes of most other containers is relative to the overall container size.

Sizing percentage-based children of a linear container with automatic positioning

When Flex sizes children of a container that uses automatic positioning to lay out children in a single direction, Flex does the following:

1 Determines the size of the viewable area of the parent container, and uses those dimensions for sizing calculations. The viewable area is the part of the container that is being displayed and can contain child components, text, images, or other content. For more information on calculating the size of containers, see "Determining component size" on page 367.

2 Determines the desired sizes of container children with percentage-based sizes as a percentage of the viewable area of the container, minus any padding and gaps between the children.

3 Reserves space for all children with explicit or default sizes.

4 If available space (parent container size minus all reserved space, including borders, padding, and gaps) cannot accommodate the percentage requests, divides the available space in proportion to the specified percentages.

5 If a minimum or maximum height or width specification conflicts with a calculated value, uses the minimum or maximum value, and recalculates all other percentage-based components based on the reduced available space.

6 Rounds the size down to the next integer.

The following examples show how the requested percentage can differ from the size when the component is laid out:

- Suppose that 50% of a SkinnableContainer is available after reserving space for all explicit-sized and default-sized children, and for all gaps and padding. If one component requests 20% of the parent, and another component requests 60%, the first component is sized to 12.5% (20 /(20+ 60) * 50%) of the parent container and the second component is sized to 37.5% of the parent container.
- If any component requests 100% of its parent application container's space, it occupies all the container *except* for the application's padding.

Example: Using percentage-based children

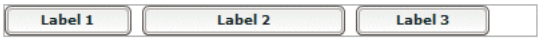
The following example specifies percentage-based sizes for the first two of three buttons in a BorderContainer container:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildren.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:BorderContainer width="400" height="25">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button id="b1" label="Label 1" width="25%"/>
        <s:Button id="b2" label="Label 2" width="40%"/>
        <s:Button id="b3" label="Label 3"/>
    </s:BorderContainer>
</s:Application>
```

In this example, the default width of the third button is 70 pixels. The BorderContainer container has no padding by default, but has a one-pixel wide border on both sides. Therefore, of the 400 pixel width, only 398 are available for laying out children. The HorizontalLayout class uses a 6-pixel horizontal gap between each component.

The first button requests 25% of the available 398 pixels of space, or 100 pixels (All calculated values are to the nearest pixel). The second button requests 40%, or 159 pixels. There is still unused space to the right of the third button.

Flex draws the following application:



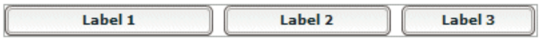
Now change the percentage values requested to 50% and 40%, respectively:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildren5040.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:BorderContainer width="400" height="25">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button id="b1" label="Label 1" width="50%"/>
        <s:Button id="b2" label="Label 2" width="40%"/>
        <s:Button id="b3" label="Label 3"/>
    </s:BorderContainer>
</s:Application>
```

In this example, the first button requests 50% of the available space, or 200 pixels. The second button still requests 40%, or 160 pixels, for a total of 360 pixels. However, the BorderContainer container only has 318 pixels free after reserving 70 pixels for the default-width button, 2 pixels for the borders, and 12 pixels for the gaps between components.

Flex divides the available space proportionally between the two buttons, giving .5/(.5 + .4) * 318 = 176 pixels, to the first button and .4/(.5 + .4) * 318 = 140 pixels, to the second button.

Flex draws the following application:



Sizing percentage-based children of a container with absolute positioning

Spark containers with a layout of BasicLayout, or MX Canvas, MX Panel, and MX Application containers can use absolute positioning. When Flex sizes children of a container that uses absolute positioning, it does the following:

1 Determines the viewable area of the parent container, and uses the corresponding dimensions as the container dimensions for sizing calculations. For more information on calculating the size of containers, see "Determining component size" on page 367.

2 Determines the sizes of children with percentage-based sizes by multiplying the decimal value by the container dimension minus the position of the control in the dimension's direction. For example, if you specify `x="10"` and `width="100%"` for a child, the child size extends only to the edge of the viewable area, not beyond.

   ***Note:** The Spark BasicLayout class always calculates the percentage based on the size of the container, regardless of the position of the child component in the container.*

   Because controls can overlay other controls or padding, the sizing calculations do not consider padding or any other children when determining the size of a child.

3 If a minimum or maximum height or width specification conflicts with a calculated value, uses the minimum or maximum value.

4 For MX containers, round the size down to the closest integer.

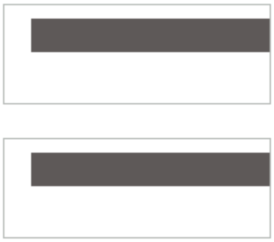
   For Spark containers, round the size up or down to the closest integer.

Example: Using percentage-based children with absolute positioning

The following code shows the percentage-based sizing behavior with absolute positioning:

```
<?xml version="1.0"?>
<!-- components\PercentSizeAbsPosit.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout gap="25"/>
    </s:layout>
    <s:BorderContainer
        width="200" height="75">
        <s:layout>
            <s:BasicLayout/>
        </s:layout>
        <s:BorderContainer
            x="20" y="10"
            width="100%" height="25"
            backgroundColor="#666666"/>
    </s:BorderContainer>
    <s:BorderContainer
        width="200" height="75">
        <s:layout>
            <s:BasicLayout/>
        </s:layout>
        <s:BorderContainer
            left="20" top="10"
            width="100%" height="25"
            backgroundColor="#666666"/>
    </s:BorderContainer>
</s:Application>
```

Flex draws the following application:



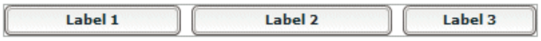
Using minimum or maximum dimensions

You can also use the `minWidth`, `minHeight`, `maxWidth`, and `maxHeight` properties with a percentage-based component to constrain its size. Consider the following example:

```
<?xml version="1.0"?>
<!-- components\PercentChildrenMin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:BorderContainer id="bc" width="400" height="25">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button id="b1" label="Label 1" width="50%"/>
        <s:Button id="b2" label="Label 2" width="40%" minWidth="150"/>
        <s:Button id="b3" label="Label 3"/>
    </s:BorderContainer>

    <s:Form>
        <s:FormItem label="Container:">
            <s:Label text="{bc.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #1:">
            <s:Label text="{b1.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #2:">
            <s:Label text="{b2.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #3:">
            <s:Label text="{b3.width}"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

To determine the widths of the percentage-based button sizes, Flex first determines the sizes as described in the second example in “Example: Using percentage-based children” on page 374, which results in requested values of 176 for the first button and 140 for the second button. However, the minimum width of the second button is 150, so Flex sets its size to 150 pixels, and reduces the size of the first button to occupy the remaining available space, which results in a width of 166 pixels.

Flex draws the following application:



Setting the application container size

When you size an application, you often start by setting the size of the Application or Application container. The application containers determine the boundaries of your application in the Adobe® Flash® Player or Adobe® AIR™.

If you are using Adobe® Flash® Builder™, an HTML wrapper page is generated automatically. The `width` and `height` properties specified by the application container are used to set the width and height of the `<object>` and `<embed>` tags in the HTML wrapper page. Those numbers determine the portion of the HTML page that is allocated to the Adobe Flash plug-in.

If you are not autogenerating the HTML wrapper, set the application containers `width` and `height` properties to 100%. That way, the application scales to fit the space that is allocated to the Flash plug-in.

Set the application container size as the following example shows:

```
<?xml version="1.0"?>
<!-- components\AppExplicit.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="100"
    width="150">

        <!-- Application children go here. -->
</s:Application>
```

In this example, you set the application container size to 100 by 150 pixels. Anything in the application larger than this window is clipped at the window boundaries. Therefore, if you define a 200 pixel by 200 pixel DataGrid control, it is clipped.

If you are using the Spark Application container, you can modify the container's skin to add scroll bars. For more information, see "Using scroll bars with the application container" on page 395. If you are using the MX Application container, it automatically displays scroll bars.

For more information on sizing the Application container, see "Application containers" on page 393.

Using padding and custom gaps

There may be situations where you want your containers to have padding around the edges. Some containers, such as the MX Application container, have padding by default; others, such as the Spark Application container, have padding values of 0 by default.

Use the `paddingTop`, `paddingBottom`, `paddingRight`,and `paddingLeft` style properties to set the padding around the content area of a MX container. To set the padding for a Spark container, you can either use the padding properties of the layout classes, such as HorizontalLayout or VerticalLayout, or define a container skin.

Also, MX containers and Spark layouts define gaps between children, which you might want to change from the default values. If your application has nonzero padding and gaps, Flex reserves the necessary pixels before it sizes any percentage-based components.

Consider the following example:

```
<?xml version="1.0"?>
<!-- components\PadContainer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:BorderContainer width="400" height="25">
        <s:layout>
            <s:HorizontalLayout
                paddingLeft="5"
                paddingRight="5"
                gap="15"/>
        </s:layout>

        <s:Button id="b1"
            label="Label 1"
            width="75%"/>
        <s:Button id="b2"
            label="Label 2"
            minWidth="120"/>
        <s:Button id="b3"
            label="Label 3"/>
    </s:BorderContainer>
    <s:Form>
        <s:FormItem label="Button #1:">
            <s:Label text="{b1.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #2:">
            <s:Label text="{b2.width}"/>
        </s:FormItem>
        <s:FormItem label="Button #3:">
            <s:Label text="{b3.width}"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

The left and right padding of the container is set to 5 pixels, and the gap between children is 15 pixels. The default width of the third button is 70 pixels. That leaves 288 pixels for the two percentage-based components. The second button requires its minimum size of 120 pixels, leaving 168 pixels available for the first button.

Positioning components

The position of a component is defined by the x and y coordinates of its top, left corner in its parent container. For most containers, Flex automatically determines the location of the component in its parent container. This is called *automatic positioning*.

Some containers let you explicitly specify the x and y coordinates of the component in it parent container. This is called *absolute positioning*. These containers include the following:

- A Spark container using the BasicLayout class
- The MX Canvas container
- The MX Application and Panel containers with the `layout` property set to `absolute`

Automatic positioning

With automatic positioning, Flex positions the container children according to the container's layout rules, such as the layout direction, the container padding, and the gaps between children of that container.

For containers that use automatic positioning, setting the `x` or `y` property directly on a child or calling the child's `move()` method has no effect, or only a temporary effect. This is because the container recalculates the child's position and does not use the specified value. You can, however, specify absolute positions for the children of these containers under some circumstances; for more information, see below.

You can control aspects of the layout by specifying container properties; for details on the properties, see the property descriptions for the container in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. You also control the layout by controlling component sizes and by using techniques such as adding spacers.

Using the Spacer control to control layout

Flex includes a Spacer control that helps you lay out children within a parent container. The Spacer control is invisible, but it does allocate space within its parent.

In the following example, you use a percentage-based Spacer control to push the Button control to the right so that it is aligned with the right edge of the HBox container:

```
<?xml version="1.0"?>
<!-- components\SpacerHBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            [Embed(source="assets/flexlogo.jpg")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </fx:Script>

    <s:SkinnableContainer width="400">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <mx:Image source="{imgCls}"/>
        <s:Label  text="Company XYZ"/>
        <mx:Spacer width="100%"/>
        <s:Button label="Close"/>
    </s:SkinnableContainer>
</s:Application>
```

In this example, the Spacer control is the only percentage-based component in the parent container. Flex sizes the Spacer control to occupy all available space in the container that is not required for other components. By expanding the Spacer control, Flex pushes the Button control to the right edge of the container.

You can use all sizing and positioning properties with the Spacer control, such as `width`, `height`, `maxWidth`, `maxHeight`, `minWidth`, and `minHeight`.

Disabling automatic positioning temporarily

You can use effects, such as the Move and Zoom effects, to modify the size or position of a child in response to a user action. For example, you might define a child so that when the user selects it, the child moves to the top of the container and doubles in size. These effects modify the `x` and `y` properties of the child as part of the effect. Similarly, you might want to change the position of a control by changing its *x* or *y* coordinate value, for example, in response to a button click.

Containers that use automatic positioning ignore the values of the `x` and `y` properties of their children during a layout update. Therefore, the layout update cancels any modifications to the `x` and `y` properties performed by the effect, and the child does not remain in its new location.

You can prevent Flex from performing automatic positioning updates that conflict with the requested action of your application by setting the `autoLayout` property of a container to `false`. Setting this property to `false` prevents Flex from laying out the container's contents when a child moves or resizes. The default value is `true`, which enables Flex to update layouts.

For MX containers only, even when you set the `autoLayout` property of a container to `false`, Flex updates the layout when you add or remove a child. Application initialization, deferred instantiation, and the `<mx:Repeater>` tag add or remove children, so layout updates always occur during these processes, regardless of the value of the `autoLayout` property. Therefore, during container initialization, Flex defines the initial layout of the container children regardless of the value of the `autoLayout` property.

The following example disables layout updates for a VBox container:

```
<?xml version="1.0"?>
<!-- components\DisableVBoxLayout.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <mx:VBox id="vb1" autoLayout="false"
        width="200"
        height="200">
        <s:Button id="b1"
            label="Button 1"/>
        <s:Button id="b2"
            label="Button 2"
            click="{b2.x += 10; vb1.invalidateDisplayList();}"/>
        <s:Button id="b3"
            label="Button 3"
            creationComplete="b3.x = 100; b3.y = 75;"/>
    </mx:VBox>
</s:Application>
```

In this example, Flex initially lays out all three Button controls according to the rules of the VBox container. The `creationComplete` event listener for the third button is dispatched after the VBox control has laid out its children, but before Flex displays the buttons. Therefore, when the third button appears, it is at the *x* and *y* positions specified by the `creationComplete` listener.

After the buttons appear, Flex shifts the second button 10 pixels to the right each time a user clicks it. The button click also triggers the container to update its layout by calling the `updateDisplay()` method.

Setting the autoLayout property of a container to false prohibits Flex from updating a container's layout after a child moves or resizes, so you should set it to false only when necessary. You should always test your application with the autoLayout property set to the default value of true, and set it to false only as necessary for the specific container and specific actions of the children in that container.

For more information on effects, see "Introduction to effects" on page 1784.

Preventing layout of hidden controls

By default, Flex lays out and reserves space for all components, including hidden components, but it does not display the hidden controls. A hidden component is one with its visible property set to false. You see blank spots where the hidden controls should appear when you make them visible. In place of the hidden controls, you see their container's background.

However, you can prevent Flex from considering the child component when it lays out the container's other children by setting the child component's includeInLayout property of the component to false.

When a component's includeInLayout property is false, Flex does not include it in the layout calculations for other components. In other words, Flex does not reserve space for the component, but still draws it. As a result, the component can appear underneath the components that follow it in the layout order. To prevent Flex from drawing the component, you must also set its visible property to false.

The following example shows the effects of the includeInLayout and visible properties. It lets you toggle each of these properties independently on the middle of three Panel controls in a SkinnableContainer:

```
<?xml version="1.0"?>
<!-- components\HiddenBoxLayout.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="500">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:SkinnableContainer>
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Panel id="p1"
            title="Panel 1"/>
        <s:Panel id="p2"
            title="Panel 2"/>
        <s:Panel id="p3"
            title="Panel 3"/>
    </s:SkinnableContainer>
    <s:SkinnableContainer>
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button label="Toggle Panel 2 Visible"
            click="{p2.visible=!p2.visible;}"/>
        <s:Button label="Toggle Panel 2 in Layout"
            click="{p2.includeInLayout=!p2.includeInLayout;}"/>
    </s:SkinnableContainer>
</s:Application>
```

Run this application and click the buttons to see the results of different combinations of `visible` and `includeInLayout` properties. The example shows the following behaviors:

- If you include the second Panel container in the layout but make it invisible, Flex reserves space for it.
- If you exclude the second Panel container from the layout, the SkinnableContainer resizes and the third Panel container moves up. If you then include the second Panel container in the layout, the SkinnableContainer resizes again, and the third Panel container moves down.
- If you exclude the second Panel container from the layout but make it visible, Flex still draws it, but does not consider it in laying out the third Panel container, so the two panels overlap.

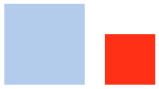
Absolute positioning

With absolute positioning, you specify the position of the child by setting its `x` and `y` properties, or you specify a constraint-based layout; otherwise, Flex places the child at position 0,0 of the parent container. When you specify the *x* and *y* coordinates, Flex repositions the controls only when you change the property values.

The following example uses absolute positioning to place two Spark BorderContainer containers inside a SkinnableContainer. Absolute positioning is the default layout of a SkinnableContainer:

```
<?xml version="1.0"?>
<!-- components\AbsoluteLayout.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:SkinnableContainer
        width="100" height="100">
        <s:BorderContainer id="b1"
            width="80" height="80"
            x="20" y="20"
            backgroundColor="#A9C0E7">
        </s:BorderContainer>
        <s:BorderContainer id="b2"
            width="50" height="50"
            x="120" y="50"
            backgroundColor="#FF0000">
        </s:BorderContainer>
    </s:SkinnableContainer>
</s:Application>
```

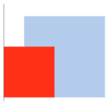
This example produces the following image:



When you use absolute positioning, you have full control over the locations of the container's children. This lets you overlap components. The following example moves the second BorderContainer container so that it partially overlaps the first.

```
<?xml version="1.0"?>
<!-- components\AbsoluteLayoutOverlap.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:SkinnableContainer
        width="100" height="100">
        <s:BorderContainer id="b1"
            width="80" height="80"
            x="20" y="20"
            backgroundColor="#A9C0E7">
        </s:BorderContainer>

        <s:BorderContainer id="b2"
            width="50" height="50"
            x="0" y="50"
            backgroundColor="#FF0000">
        </s:BorderContainer>
    </s:SkinnableContainer >
</s:Application>
```

This example produces the following image:



***Note:** If you use percentage-based sizing for the children of a control that uses absolute positioning, the percentage-based components resize when the parent container resizes, and the result may include unwanted overlapping of controls.*

Using constraints to control component layout

You can manage a child component's size and position simultaneously by using constraint-based layout, or by using constraint rows and columns. Constraint-based layout lets you anchor the sides or center of a component to positions relative to the viewable region of the component's container. The *viewable region* is the part of the component that is being displayed, and it can contain child controls, text, images, or other contents.

Constraint rows and columns let you subdivide a container into vertical and horizontal constraint regions to control the size and positioning of child components with respect to each other and within the parent container.

Creating a constraint-based layout

You can use constraint-based layout to determine the position and size of the immediate children of any container that supports absolute positioning. With constraint-based layout, you can do the following:

- Anchor one or more edges of a component at a pixel offset from the corresponding edge of its container's viewable region. The anchored child edge stays at the same distance from the parent edge when the container resizes. If you anchor both edges in a dimension, such as top and bottom, the component resizes if the container resizes.
- Anchor the child's horizontal or vertical center (or both) at a pixel offset from the center of the container's viewable region. The child does not resize in the specified dimension unless you also use percentage-based sizing.
- Anchor the baseline of a component at a pixel offset from the top edge of its parent container.

You can specify a constraint-based layout for any Flex framework component (that is, any component that extends the UIComponent class). The following rules specify how to position and size components by using constraint-based layout:

- Place the component in any Spark container that uses BasicLayout, in a MX Canvas container, or in a MX Application or MX Panel container with the `layout` property set to `absolute`.
- Specify the constraints by using the `baseline`, `top`, `bottom`, `left`, `right`, `horizontalCenter`, or `verticalCenter` properties of UIComponent and GraphicElement.

  ***Note:** In previous releases of Flex, the baseline, top, bottom, left, right, horizontalCenter, or verticalCenter properties were implemented as styles. You can still use them as styles in this release.*

  The `top`, `bottom`, `left`, and `right` properties specify the distances between the component sides and the corresponding container sides.

  The `baseline` constraint specifies the distance between the baseline position of a component and the upper edge of its parent container. Every component calculates its baseline position as the y-coordinate of the baseline of the first line of text of the component. The baseline of a UIComponent object that does not contain any text is calculated as if the UIComponent object contained a UITextField object that uses the component's styles, and the top of the UITextField object coincides with the component's top.

  The `horizontalCenter` and `verticalCenter` properties specify distance between the component's center point and the container's center, in the specified direction; a negative number moves the component left or up from the center.

  The following example anchors the Form control's left and right sides 20 pixels from its container's sides:

  ```
  <mx:Form id="myForm" left="20" right="20"/>
  ```

- Do not specify a `top` or `bottom` property with a `verticalCenter` property; the `verticalCenter` value overrides the other properties. Similarly, do not specify a `left` or `right` property with a `horizontalCenter` property.
- A size determined by constraint-based layout overrides any explicit or percentage-based size specifications. If you specify `left` and `right` constraints, for example, the resulting constraint-based width overrides any width set by a `width` or `percentWidth` property.

Precedence rules for constraint-based components

- If you specify a single edge constraint (`left, right, top,` or `bottom`) without any other sizing or positioning parameter, the component size is the default size and its position is determined by the constraint value. If you specify a size parameter (width or height), the size is determined by that parameter.
- If you specify a pair of constraints (`left-right` or `top-bottom`), the size and position of the component is determined by those constraint values. If you also specify a center constraint (`horizontalCenter` or `verticalCenter`), the size of the component is calculated from the edge constraints and its position is determined by the center constraint value.
- Component size determined by a pair of constraint-based layout properties (`left-right` or `top-bottom`) overrides any explicit or percentage-based size specifications. For example, if you specify both `left` and `right` constraints, the calculated constraint-based width overrides the width set by a `width` or `percentWidth` property.
- Edge constraints override `baseline` constraints.

Example: Using constraint-based layout for a form

The following example code shows how you can use constraint-based layout for a form. In this example, the Form control uses a constraint-based layout to position its top just inside the canvas padding. The form left and right edges are 20 pixels from the outer SkinnableContainer container's left and right edges. The second, inner, SkinnableContainer that contains the buttons uses a constraint-based layout to place itself 20 pixels from the right edge and 10 pixels from the bottom edge of the outer SkinnableContainer container.

If you change the size of your browser, stand-alone Flash Player, or AIR application, you can see the effects of dynamically resizing the application container on the Form layout. The form and the buttons overlap as the application grows smaller, for example.

```
<?xml version="1.0"?>
<!-- components\ConstraintLayout.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:SkinnableContainer width="100%" height="100%">
         <!-- Anchor the top of the form at the top of the canvas.
            Anchor the form sides 20 pixels from the canvas sides. -->
        <s:Form id="myForm"
                backgroundColor="#DDDDDD"
                top="0"
                left="20"
                right="20">

            <s:FormItem label="Product:" width="100%">
                <!-- Specify a fixed width to keep the ComboBox control from
                    resizing as you change the application size. -->
                <s:ComboBox width="200"/>
            </s:FormItem>
            <s:FormItem label="User" width="100%">
                <s:ComboBox width="200"/>
            </s:FormItem>
            <s:FormItem label="Date">
                <mx:DateField/>
            </s:FormItem>
            <s:FormItem width="100%"
                    label="Hours:">
                <s:TextInput width="75"/>
            </s:FormItem>
            <s:FormItem width="100%"
                    label="Minutes:">
                <s:TextInput width="75"/>
            </s:FormItem>
        </s:Form>
        <!-- Anchor the box with the buttons 20 pixels from the canvas
            right edge and 10 pixels from the bottom. -->
        <s:SkinnableContainer id="okCancelBox"
                right="20"
                bottom="10">
            <s:layout>
                <s:HorizontalLayout/>
            </s:layout>
            <s:Button label="OK"/>
            <s:Button label="Cancel"/>
        </s:SkinnableContainer>
    </s:SkinnableContainer>
</s:Application>
```

Using constraint rows and columns with MX containers and the Spark FormLayout class

You can subdivide a container that supports absolute positioning into vertical and horizontal constraint regions to control the size and positioning of child components with respect to each other, or with respect to the parent container.

***Note:** Constraint rows and columns are only supported by the MX containers, and the Spark FormLayout class.*

You define the horizontal and vertical constraint regions of a container by using the `constraintRows` and `constraintColumns` properties. These properties contain Arrays of constraint objects that partition the container horizontally (ConstraintColumn objects) and vertically (ConstraintRow objects). ConstraintRow objects are laid out in the order they are defined, from top to bottom in their container; ConstraintColumn objects are laid out from left to right in the order they are defined.

The following example shows a `Canvas` container partitioned into two vertical regions and two horizontal regions. The first constraint column occupies 212 pixels from the leftmost edge of the `Canvas`. The second constraint column occupies 100% of the remaining Canvas width. The rows in this example occupy 80% and 20% of the Canvas container’s height from top to bottom, respectively.

```
<?xml version="1.0"?>
<!-- constraints\BasicRowColumn.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:Canvas>
        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="212"/>
            <mx:ConstraintColumn id="col2" width="100%"/>
        </mx:constraintColumns>
        <mx:constraintRows>
            <mx:ConstraintRow id="row1" height="80%"/>
            <mx:ConstraintRow id="row2" height="20%"/>
        </mx:constraintRows>
    <!-- Position child components based on
        the constraint columns and rows. -->
    </mx:Canvas>
</s:Application>
```

Constraint rows and columns do not have to occupy 100% of the available area in a container. The following example shows a single constraint column that occupies 20% of the Canvas width; 80% of the container is unallocated:

```
<?xml version="1.0"?>
<!-- constraints\BasicColumn_20Percent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:Canvas>
        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="20%"/>
        </mx:constraintColumns>
    </mx:Canvas>
    <!-- Position child components based on
        the constraint column. -->
</s:Application>
```

Creating constraint rows and columns

Constraint columns and rows have three sizing options: fixed, percent, and content. These options dictate the amount of space that the constraint region occupies in the container. As child components are added to or removed from the parent container, the space allocated to each ConstraintColumn and ConstraintRow instance is computed according to its sizing option.

- *Fixed size* means the space allocated to the constraint region is a fixed pixel size. In the following example, you set the fixed with of a ConstraintColumn instance to 100 pixels:

  ```
  <mx:ConstraintColumn id="col1" width="100"/>
  ```

  As the parent container grows or shrinks, the ConstraintColumn instance remains 100 pixels wide.

- *Percent size* means that the space allocated to the constraint row or column is calculated as a percentage of the space remaining in the parent container after the space allocated to fixed and content size child objects has been deducted from the available space.

  In the following example, you set the width of a ConstraintColumn instance to 80%:

  ```
  <mx:ConstraintColumn  id="col1" width="80%"/>
  ```

  As the parent container grows or shrinks, the ConstraintColumn always takes up 80% of the available width.

  A best practice in specifying percent constraints is to ensure that the sum of all percent constraints is less than or equal to 100%. However, if the total value of percent specifications is greater than 100%, the actual allocated percentages are calculated so that the proportional values for all constraints total 100%. For example, if the percentages for two constraint objects are specified as 100% and 50%, the values are adjusted to 66.6% and 33.3% (two-thirds for the first value and one-third for the second).

- *Content size* (default) means that the space allocated to the region is dictated by the size of the child objects in that space. As the size of the content changes, so does the size of the region. Content sizing is the default when you do not specify either fixed or percentage sizing parameters.

  In the following example, you specify content size by omitting any explicit width setting:

  ```
  <mx:ConstraintColumn  id="col1"/>
  ```

  The width of this ConstraintColumn is determined by the width of its largest child. When children span multiple content sized constraint rows or constraint columns, Flex divides the space consumed by the children among the rows and columns.

For the ConstraintColumn class, you can also use the `maxWidth` and `minWidth` properties to limit the width of the column. For the ConstraintRow class, you can use the `maxHeight` and `minHeight` properties to limit the height of the row. Minimum and maximum sizes for constraint columns and rows limit how much the constraint regions grow or shrink when you resize their parent containers. If the parent container with a constraint region shrinks to less than the minimum size for that region when you resize the container, scroll bars appear to show clipped content.

***Note:** Minimum and maximum limits are only applicable to percentage and content sized constraint regions. For fixed size constraint regions, minimum and maximum values, if specified, are ignored.*

Positioning child components based on constraint rows and constraint columns

Anchor a child component to a constraint row or constraint column by prepending the constraint region's ID to any of the child's constraint parameters. For example, if the ID of a ConstraintColumn is "col1", you can specify a set of child constraints as `left="col1:10"`, `right="col1:30"`, `horizontalCenter="col1:0"`.

If you do not qualify constraint parameters (`left, right, top`, and `bottom`) a constraint region ID, the component is constrained relative to the edges of its parent container. Components can occupy a single constraint region (row or column) or can span multiple regions.

The following example uses constraint rows and constraint columns to position three Button controls in a Canvas container:

```
<?xml version="1.0"?>
<!-- constraints\ConstrainButtons.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:Canvas id="myCanvas" backgroundColor="0x6699FF">
        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="100"/>
            <mx:ConstraintColumn id="col2" width="100"/>
        </mx:constraintColumns>
        <mx:constraintRows>
            <mx:ConstraintRow id="row1" height="100"/>
            <mx:ConstraintRow id="row2" height="100"/>
        </mx:constraintRows>
        <s:Button label="Button 1"
            top="row1:10" bottom="row1:10"
            left="10"/>
        <s:Button label="Button 2"
            left="col2:10" right="col2:10"/>
        <s:Button label="Button 3"
            top="row2:10" bottom="row2:10"
            left="col1:10" right="10"/>
    </mx:Canvas>
    <s:Label text="canvas width:{myCanvas.width}"/>
    <s:Label text="canvas height:{myCanvas.height}"/>
</s:Application>
```

The next example defines the constraint rows and columns by using percentages. As you resize the application, the Button controls resize accordingly.

```
<?xml version="1.0"?>
<!-- constraints\ConstrainButtonsPercent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:Canvas id="myCanvas"
        backgroundColor="0x6699FF"
        width="100%" height="100%">
        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="30%"/>
            <mx:ConstraintColumn id="col2" width="40%"/>
            <mx:ConstraintColumn id="col3" width="30%"/>
        </mx:constraintColumns>
        <mx:constraintRows>
            <mx:ConstraintRow id="row1" height="35%"/>
            <mx:ConstraintRow id="row2" height="55%"/>
        </mx:constraintRows>
        <s:Button label="Button 1"
            top="row1:10" bottom="row2:10"
            left="10"/>
        <s:Button label="Button 2"
            left="col2:10" right="col3:10"/>
        <s:Button label="Button 3"
            top="row2:10" bottom="row2:10"
            left="col3:10" right="col3:10"/>
    </mx:Canvas>
    <s:Label text="canvas width:{myCanvas.width}"/>
    <s:Label text="canvas height:{myCanvas.height}"/>
</s:Application>
```

While you can specify any combination of qualified and unqualified constraints, some constraint properties may be overridden. The priority of sizing and positioning constraints are as follows:

1 Center constraint specifications override all other constraint values when determining the position of a control.

2 Next, left edge and top positions are determined.

3 Finally, right edge and bottom positions are calculated to best fit the component.

The following table defines the behavior of constrained components when they are contained in a single constraint region. *Edge 1* is the first specified edge constraint for a child component (`left`, `right`, `top`, or `bottom`). *Edge 2* is the second specified edge constraint of a pair (`left` and `right`, `top` and `bottom`). *Size* is an explicit size for a child component (`width`, `height`). *Center* is the positioning constraint to center the child object (`horizontalCenter` or `verticalCenter`).

Constraint Parameters				Behavior	
Edge 1	**Edge 2**	**Size**	**Center**	**Size**	**Position**
x				Default component size	Relative to specified edge constraint
x	x			Calculated from specified edge constraints	Relative to specified edge constraints
x	x	x		Determined from specified edge constraints. Explicit size is overridden	Relative to specified edge constraints
x		x		Specified size	Relative to specified edge
x			x	Default component size	Centered in constraint region; edge constraint is ignored
x	x		x	Calculated from edge constraints	Centered in constraint region
x		x	x	Explicit size	Centered in constraint region; single edge constraint is ignored
			x	Default size of component	Centered in constraint region

Using Baseline property

The baseline property defines the offset between the baseline defined by the constraint row (or the top of the container) and the baseline position of the element.

The baseline property can be specified as a number or can be specified with respect to the max ascent of the row.

If the baseline value on a row is only a number, the elements contained by the row are positioned that many pixels below the top of the container. If the baseline value is defined with respect to the max ascent of the row, then it is defined in the format: "maxAscent:x". The default value is "maxAscent:0".

The following code example defines four labels in a group, where each label is constrained with a baseline alignment to row1 and maxAscent:5.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\layouts\spark\SparkFormItemLayoutBaselineAlignment.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Group y="200" x="30">
        <s:layout>
            <s:FormItemLayout>
                <s:constraintRows>
                    <s:ConstraintRow id="row1" />
                    <s:ConstraintRow id="row2" baseline="maxAscent:5"/>
                    <s:ConstraintRow id="row3" baseline="30"/>
                </s:constraintRows>
            </s:FormItemLayout>
        </s:layout>
        <s:Label fontSize="20" text="hello!" baseline="row1:10" x="38" color="blue"/>
        <s:Label fontSize="25" text="hello!" baseline="row1:0" x="89" color="#FF9933"/>
        <s:Label fontSize="35" text="hello!" baseline="row1:0" x="149" color="green"/>
        <s:Label fontSize="50" text="hello!" baseline="row1:0" x="234" />
    </s:Group>
</s:Application>
```

Application containers

Adobe® Flex® defines the Spark Application and MX Application containers that let you start adding content to your application without having to explicitly define another container.

The application containers support an application preloader that uses a progress bar to show the download progress of an application SWF file. You can override the default progress bar to define your own custom progress bar. For more information, see “Showing the download progress of an application” on page 408.

About the Application container

The first tag in an MXML application is either the `<s:Application>` tag for the Spark application container, or the `<mx:Application>` tag for an MX application container. The application container then becomes the default container for any content that you add to your application.

The application object, of type spark.components.Application or mx.core.Application, is the default scope for any ActionScript code in the file. Therefore, the ActionScript keyword `this` refers to the application object.

You also use the application container to define the initial size of the application. By default, the application sets its height to 375 pixels and its width to 500 pixels.

Although you might find it convenient to use an application container as the only container in your application, usually you explicitly define at least one more container before you add any controls to your application. For example, you might use a Panel container as the first child container of the application.

The application containers have the following default layout characteristics:

Characteristic	Spark Application container	MX Application container
Default size	375 pixels high and 500 pixels wide in the Standalone Flash Player, and all available space in a browser.	375 pixels high and 500 pixels wide in the Standalone Flash Player, and all available space in a browser.
Child layout	BasicLayout, meaning that you have to explicitly size and position all children.	Vertical column arrangement of children centered in the Application container.
Default padding	0 pixels.	24 pixels for the `paddingTop`, `paddingBottom`, `paddingLeft`, and `paddingRight` properties.
Scroll bars	Added by skinning the container.	Built into the container.

Sizing an application container and its children

You can set the height and width of an application container by using explicit pixel values or by using percentage values, where the percentage values are relative to the size of the browser window. By default, the application container sets its height to 375 pixels and its width to 500 pixels in the Standalone Flash Player, and sizes itself to use all available space in a browser.

The following example sets the size of the Application container to one-half of the width and height of the browser window:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="50%" width="50%">
    ...
</Application>
```

The advantage of using percentages to specify the size is that Flex can resize your application as the user resizes the browser window. Flex maintains the application container size as a percentage of the browser window as the user resizes it.

If you set the `width` and `height` properties of the child components to percentage values, your components can also resize as your application resizes, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppSizePercent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="100%" height="100%">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Main Application" width="100%" height="100%">
        <mx:HDividedBox width="100%" height="100%">
            <s:TextArea text="TA1" width="50%" height="100%"/>
            <mx:VDividedBox width="50%" height="100%">
                <s:TextArea text="TA2" width="100%" height="75%"/>
                <s:TextArea text="TA3" width="100%" height="75%"/>
            </mx:VDividedBox>
        </mx:HDividedBox>
    </s:Panel>
</s:Application>
```

The following example uses explicit pixel values to size the application container:

```
<?xml version="1.0"?>
<!-- containers\application\AppSizePixel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="200" height="150">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:Panel title="Main Application">
        <mx:TextInput id="mytext" text="Hello"/>
        <mx:Button id="mybutton" label="Get Weather"/>
    </mx:Panel>
</s:Application>
```

If you want to set a child container to fill the entire application container, the easiest method is to set the child's `width` and `height` properties to 100% in MXML. In ActionScript, set the `percentWidth` and `percentHeight` properties to 100.

Because the Spark Application container defines 0 pixels of padding around its content area, a child sized to 100% fills the entire area of the container. In the following example, the child VBox container fills the entire application container:

```
<?xml version="1.0"?>
<!-- containers\application\AppNoPadding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="100%" height="100%">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <mx:VBox width="100%" height="100%" backgroundColor="#A9C0E7">
        <!-- ... -->
    </mx:VBox>
</s:Application>
```

The MX Application container defines 24 pixels of padding around all four sides of its content area. Therefore, if you want a child to occupy the entire area of the Application container, set the container's padding properties (`paddingTop`, `paddingBottom`, `paddingLeft`, `paddingRight`) to 0.

Using scroll bars with the application container

The MX Application container has built in support for scroll bars. Therefore, if the children of the MX Application container are sized or positioned such that some or all the component is outside the visible area of the MX Application container, Flex adds scroll bars to the container.

In the following example, the VBox container in the MX Application container is larger than the available space within the Application container, which results in scroll bars:

```
<?xml version="1.0"?>
<!-- containers\application\AppVBoxSizeScroll.mxml -->
<mx:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="100" height="100">
    <mx:VBox width="200" height="200" backgroundColor="#A9C0E7">
        <!-- ... -->
    </mx:VBox>
</mx:Application>
```

To add scroll bars to the Spark Application container, define a skin that supports scroll bars. The following example shows a skin in the file MyAppSkin.mxml. This skin modifies the default skin for the Spark Application container, spark.skins.default.ApplicationSkin, to add the Spark Scroller component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\application\mySkins\MyAppSkin.mxml -->
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Metadata>
        [HostComponent("spark.components.Application")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>

    <!-- fill -->
    <s:Rect id="backgroundRect" left="0" right="0" top="0" bottom="0">
        <s:fill>
            <s:SolidColor color="0xFFFFFF" />
        </s:fill>
    </s:Rect>

    <s:Scroller height="100%" width="100%">
        <s:Group id="contentGroup" left="0" right="0" top="0" bottom="0" />
    </s:Scroller>
</s:Skin>
```

The following application uses this skin:

```
<?xml version="1.0"?>
<!-- containers\application\AppSparkScroll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="150" height="150"
    skinClass="mySkins.MyAppSkin">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:VBox width="200" height="200" backgroundColor="#A9C0E7">
        <!-- ... -->
    </mx:VBox>

    <mx:VBox width="200" height="200" backgroundColor="#000000">
        <!-- ... -->
    </mx:VBox>
</s:Application>
```

For more information, see “Adding scroll bars to Spark containers” on page 1637.

Overriding the default Application container styles

By default, the application containers have the following properties and style properties that define the visual aspects of the application and differ from the default container values:

Property	Spark Application default value	MX Application default value
backgroundColor	The color of the Stage area of Adobe® Flash® Player or Adobe® AIR™, which is visible during application loading and initialization. This color is also visible if the application skin does not define any other background color or image. The default value is 0xFFFFFF (white).	The color of the Stage area of Adobe® Flash® Player or Adobe® AIR™, which is visible during application loading and initialization. This color is also visible if the application background is transparent. The default value is 0xFFFFFF (white).
backgroundGradientAlphas	Not implemented; define in the container skin.	[1.0, 1.0], a fully opaque background.
backgroundGradientColors	Not implemented; define in the container skin.	undefined, which means background gradient is generated based on the backgroundColor property. By default, the backgroundColor property defines a white background.
backgroundImage	Not implemented; define in the container skin.	A gradient controlled by the backgroundGradientAlphas and backgroundGradientColors styles. The default value is mx.skins.halo.ApplicationBackground.
backgroundSize	Not implemented; define in the container skin.	100%. When you set this property at 100%, the background image takes up the entire Application container.
horizontalAlign	Not implemented; set in the layout class associated with the Application container.	Centered.
paddingBottom	0 pixels.	24 pixels.

Property	**Spark Application default value**	**MX Application default value**
`paddingLeft`	0 pixels.	24 pixels.
`paddingRight`	0 pixels.	24 pixels.
`paddingTop`	0 pixels.	24 pixels.

Changing the Spark Application container background

The Spark `Application.backgroundColor` style defines the default background color of the Application container during application loading and initialization. If the Application skin does not set a different color, the `backgroundColor` style also defines the background color of the application when it is running. The default value is 0xFFFFFF (white).

To set the background color of the Spark Application container to something other than a solid color, or to set a background image, you define a skin for the container. For example, the following skin defines a gradient fill for the background of the Application container that goes from blue to gray:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\application\mySkins\MyAppBackgroundSkin.mxml -->
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Metadata>
        [HostComponent("spark.components.Application")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>
    <!-- Define a gradient fill for the background of the Application container. -->
    <s:Rect id="backgroundRect" left="0" right="0" top="0" bottom="0">
        <s:fill>
            <s:LinearGradient>
                <s:entries>
                    <s:GradientEntry color="0x0000FF" ratio="0.00" alpha="0.5"/>
                    <s:GradientEntry color="0xCCCCCC" ratio="0.5" alpha="0.5"/>
                </s:entries>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <s:Group id="contentGroup" left="0" right="0" top="0" bottom="0" />
</s:Skin>
```

You then use the `skinClass` style property to apply the skin to the Application, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppSparkBackground.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    skinClass="mySkins.MyAppBackgroundSkin">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

</s:Application>
```

The next skin example uses the Image control to embed an image for the background of the Application container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\application\mySkins\MyAppBackgroundImageSkin.mxml -->
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Metadata>
        [HostComponent("spark.components.Application")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>

    <!-- Use an Image control to embed an image for the
            background of the Application container. -->
    <mx:Image source="@Embed(source='logo.jpg')"
        left="0" right="0" top="0" bottom="0"
        maintainAspectRatio="false" />

    <s:Group id="contentGroup" left="0" right="0" top="0" bottom="0" />
</s:Skin>
```

For more information on skinning, see “Spark Skinning” on page 1602.

Changing the MX Application container background

The Application container `backgroundGradientAlphas`, `backgroundGradientColors`, and `backgroundImage` styles control the container background. By default, these properties define an opaque gray gradient background.

You specify an image for the application background by using the `backgroundImage` property. If you set both the `backgroundImage` property and the `backgroundGradientColors` property, Flex ignores `backgroundGradientColors`.

You can specify a gradient background for the application in two ways:

1 Set the `backgroundGradientColors` property to two values, as in the following example:

```
<mx:Application xmlns="http://ns.adobe.com/mxml/2009"
    backgroundGradientColors="[0x0000FF, 0xCCCCCC]">
```

Flex calculates the gradient pattern between the two specified values.

2 Set the `backgroundColor` property to the desired value, as in the following example:

```
<mx:Application xmlns="http://ns.adobe.com/mxml/2009"
    backgroundColor="red">
```

Flex sets the gradient background to solid red.

To set a solid background to the application by using the `backgroundGradientColors` property, specify the same two values to the `backgroundGradientColors` property, as the following example shows:

```
<mx:Application xmlns="http://ns.adobe.com/mxml/2009"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">
```

This example defines a solid white background.

The `backgroundColor` property specifies the background color of the Stage area in Flash Player, which is visible during application loading and initialization, and a background gradient while the application is running. By default, the `backgroundColor` property is set to `0xFFFFFF` (white).

If you use the `backgroundGradientColors` property to set the application background, set the `backgroundColor` property to compliment the `backgroundGradientColors` property, as the following example shows:

```
<mx:Application xmlns="http://ns.adobe.com/mxml/2009"
    backgroundGradientColors="[0x0000FF, 0xCCCCCC]"
    backgroundColor="0x0000FF">
```

In this example, you use the `backgroundGradientColors` property to set a gradient pattern from a dark blue to gray, and the `backgroundColor` property to set the Stage area in Flash Player to dark blue, which is visible during application loading and initialization.

Adding a control bar area to the application container

An application control bar contains a group of controls outside of the main content area of an application container. The control bar is always visible, and cannot be scrolled like other container children.

For the MX Application container, create a control bar area by adding the ApplicationControlBar container as a child of the MX Application container. For more information and examples, see “MX ApplicationControlBar layout container” on page 582.

For the Spark Application container, use the `controlBarContent` property to define the controls that appear in the control bar area, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\application\AppSparkCB.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:controlBarContent>
        <mx:MenuBar height="100%"
            dataProvider="{menuXML}"
            labelField="@label"
            showRoot="true"/>
        <mx:HBox paddingBottom="5"
            paddingTop="5">
            <mx:ComboBox dataProvider="{cmbDP}"/>
            <mx:Spacer width="100%"/>
            <mx:TextInput id="myTI" text=""/>
            <mx:Button id="srch1"
                label="Search"
                click="Alert.show('Searching');"/>
        </mx:HBox>
    </s:controlBarContent>

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:XMLList id="menuXML">
            <fx:menuitem label="File">
                <fx:menuitem label="New" data="New"/>
                <fx:menuitem label="Open" data="Open"/>
                <fx:menuitem label="Save" data="Save"/>
                <fx:menuitem label="Exit" data="Exit"/>
            </fx:menuitem>
            <fx:menuitem label="Edit">
                <fx:menuitem label="Cut" data="Cut"/>
                <fx:menuitem label="Copy" data="Copy"/>
                <fx:menuitem label="Paste" data="Paste"/>
            </fx:menuitem>
            <fx:menuitem label="View"/>
        </fx:XMLList>

        <fx:Array id="cmbDP">
            <fx:String>Item 1</fx:String>
            <fx:String>Item 2</fx:String>
            <fx:String>Item 3</fx:String>
        </fx:Array>
    </fx:Declarations>

    <s:Button label="Button"/>
    <mx:TextArea width="300" height="200"/>
</s:Application>
```

The location and appearance of the control bar area of the Spark Application container is determined by the spark.skins.spark.ApplicationSkin class, the skin class for the Spark Application container. By default, the ApplicationSkin class defines the control bar area to appear at the top of the content area of the Application container with a grey background. Create a custom skin to change the default appearance of the control bar.

By default, the controls in the control bar area use horizontal layout. Use the `Application.controlBarLayout` property to specify a different layout, as the following example shows:

```
<s:controlBarLayout>
    <s:HorizontalLayout paddingLeft="12" gap="5"/>
</s:controlBarLayout>
```

Application events

The following events are dispatched only by the application containers:

- `applicationComplete` Dispatched after the application has been initialized, processed by the LayoutManager, and attached to the display list. This is the last event dispatched during an application's startup sequence. It is later than the application's `creationComplete` event, which gets dispatched before the preloader has been removed and the application has been attached to the display list.
- `error` Dispatched when an HTTPService call fails.

Viewing the application source code

You can use the `viewSourceURL` property of the application containers to specify a URL to the application's source code. If you set this property, Flex adds a View Source menu item to the application's context menu, which you open by right-clicking anywhere in your application. Select the View Source menu item to open the URL specified by the `viewSourceURL` property in a new browser window.

Set the `viewSourceURL` property by using MXML, not ActionScript, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppSourceURL.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    viewSourceURL="assets/AppSourceURL.txt">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:Button label="Click Me"/>
</s:Application>
```

You typically deploy your source code not as an MXML file, but as a text or HTML file. In this example, the source code is in the file AppSourceURL.txt. If you use an HTML file to represent your source code, you can add formatting and coloring to make it easier to read.

Specifying options of the Application container

You can specify several options of the application container to control your application. The following table describes these options:

Option	**Type**	**Description**
`frameRate`	Number	Specifies the frame rate of the application, in frames per second. The default value is 24.
`pageTitle`	String	Specifies a String that appears in the title bar of the browser. This property provides the same functionality as the HTML `<title>` tag.
`preloader`	String	Specifies the path of a SWC component class or ActionScript component class that defines a custom progress bar. A SWC component must be in the same directory as the MXML file or in the WEB-INF/flex/user_classes directory of your Flex web application. For more information, see “Showing the download progress of an application” on page 408.
`scriptRecursionLimit`	Number	Specifies the maximum depth of Flash Player or AIR call stack before Flash Player or AIR stops. This is essentially the stack overflow limit. The default value is 1000.
`scriptTimeLimit`	Number	Specifies the maximum duration, in seconds, that an ActionScript event listener can execute before Flash Player or AIR assumes that it has stopped processing and aborts it. The default value is 60 seconds, which is also the maximum allowable value that you can set.
`useDirectBlit`	Boolean	Specifies whether hardware acceleration is used to copy graphics to the screen (if such acceleration is available). This option only applies to applications running in the standalone Flash Player. For applications running in a browser, setting `useDirectBlit` to `true` is equivalent to setting `wmode` to `"direct"` in the HTML wrapper. For AIR applications, use the `renderMode` application descriptor tag. The default value is `false`.
`useGPU`	Boolean	Specifies whether GPU (Graphics Processing Unit) acceleration is used when drawing graphics (if such acceleration is available). This option only applies to applications running in the standalone Flash Player. For applications running in a browser, setting `useGPU` to `true` is equivalent to setting `wmode` to `"gpu"` in the HTML wrapper. For AIR applications, use the `renderMode` application descriptor tag. The default value is `false`.
`usePreloader`	Boolean	Specifies whether to disable the application preloader (`false`) or not (`true`). The default value is `true`. To use the default preloader, your application must be at least 160 pixels wide. For more information, see “Showing the download progress of an application” on page 408.

***Note:** Properties `frameRate`, `pageTitle`, `preloader`, `scriptRecursionLimit`, and `usePreloader`, cannot be set in ActionScript; they must be set in MXML code.*

About the application object

Flex compiles your application into a SWF file that contains a single application object, of type spark.components.Application or mx.core.Application. In most cases, your application has one application object. Some applications use the SWFLoader control to add more applications.

An application object has the following characteristics:

- The file defining the application object is the first file loaded by the application.
- You can refer to the application object as `mx.core.FlexGlobals.topLevelApplication` from anywhere in the application.

- If you load multiple nested applications by using the SWFLoader control, you can access the scope of each higher application in the nesting hierarchy by using `parentApplication`, `parentApplication.parentApplication`, and so on.

About documents

Every MXML file used in an application is referred to as a *document class*. For example, you define an MXML file named MyForm.mxml that contains the following code:

```
<?xml version="1.0"?>
<mx:Form xmlns:mx="library://ns.adobe.com/flex/mx">
    ...
</mx:Form>
```

MyForm.mxml is the document class and instances of MyForm.mxml are referred to as *documents*. An MXML file that includes the `<s:Application>` or `<mx:Application>` tag is the document class that defines the application object. MXML files that omit the `<s:Application>` or `<mx:Application>` tag are custom controls.

A document has the following characteristics:

- All MXML files that an application uses are document classes, including the MXML file that includes the `<s:Application>` or `<mx:Application>` tag.
- Custom ActionScript component files are document classes.
- The Flex compiler cannot compile a SWF file from a file that does not contain the `<s:Application>` or `<mx:Application>` tag.
- You can access the scope of a document's parent document by using `parentDocument`, `parentDocument.parentDocument`, and so on.
- Flex provides a `UIComponent.isDocument` property so that you can detect if any given object is a document.

Accessing document and application scopes

In your application's main MXML file, the file that contains the `<s:Application>` or `<mx:Application>` tag, you can access the methods and properties of the application object by using the `this` keyword. However, in custom ActionScript and MXML components, event listeners, or external ActionScript class files, Flex executes in the context of those components and classes, and the `this` keyword refers to the current document and not to the application object. You cannot refer to a control or method in the application from one of these child documents without specifying the location of the parent document.

Flex provides the following properties that you can use to access parent documents:

mx.core.FlexGlobals.topLevelApplication The top-level application object, regardless of where in the document tree your object executes. This object is of type spark.components.Application or mx.core.Application.

mx.core.UIComponent.parentDocument The parent document of the current document. You can use `parentDocument.parentDocument` to walk up the tree of multiple documents.

mx.core.UIComponent.parentApplication The Application object in which the current object exists. Applications can load other applications, therefore, you can access the immediate parent application by using this property. You can use `parentApplication.parentApplication` to walk up the tree of multiple applications.

Using the mx.core.FlexGlobals.topLevelApplication property

To access properties and methods of the top-level application object from anywhere in your application, you can use the `topLevelApplication` property of the FlexGlobals class. For example, you define an application that contains the method, as the following code shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppDoSomething.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;

            // Open an Alert control.
            public function doSomething():void {
                Alert.show("The doSomething() method was called.");
            }
        ]]>
    </fx:Script>

    <!-- Include the ButtonMXML.mxml component. -->
    <MyComps:ButtonMXML/>
</s:Application>
```

You can then use the `FlexGlobals.topLevelApplication` property in the ButtonMXML.mxml component to reference the `doSomething()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\myComponents\ButtonMXML.mxml -->
<s:SkinnableContainer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            // To refer to the members of the Application class,
            // you must import mx.core.Application.
            import mx.core.FlexGlobals;
        ]]>
    </fx:Script>
    <s:Button  label="MXML Button"
        click="FlexGlobals.topLevelApplication.doSomething();"/>
</s:SkinnableContainer>
```

The `topLevelApplication` property is especially useful in applications that have one or more custom MXML or ActionScript components that each use a shared set of data. At the application level, you often store shared information and provide utility functions that any of the components can access.

For example, suppose that you store the user's name at the application level and implement a utility function, `getSalutation()`, which returns the string "Hi, *userName*". The following example MyApplication.mxml file shows the application source that defines the `getSalutation()` method:

```
<?xml version="1.0"?>
<!-- containers\application\AppSalutation.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            public var userName:String="SMG";

            public function getSalutation():String {
                return "Hi, " + userName;
            }
        ]]>
    </fx:Script>
    <!-- Include the ButtonGetSalutation.mxml component. -->
    <MyComps:ButtonGetSalutation/>

</s:Application>
```

To access the `userName` and call the `getSalutation()` method in your MXML components, you can use the `topLevelApplication` property, as the following example from the MyComponent.mxml component shows:

```
<?xml version="1.0"?>
<!-- containers\application\myComponents\ButtonGetSalutation.mxml -->
<s:SkinnableContainer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="100%" height="100%">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            /* To refer to the members of the Application class,
               you must import mx.core.Application. */
            import mx.core.FlexGlobals;
        ]]>
    </fx:Script>
    <mx:Label id="myL"/>
    <s:Button label="Click Me"
        click="myL.text=FlexGlobals.topLevelApplication.getSalutation();"/>
</s:SkinnableContainer>
```

In this example, clicking the Button control executes the `getSalutation()` function to populate the Label control.

Using the parentDocument property

To access the parent document of an object, you can use the parentDocument property. The parent document is the object that contains the current object. All classes that inherit from the UIComponent class have a `parentDocument` property.

In the following example, the application references the custom AccChildObject.mxml component:

```
<?xml version="1.0"?>
<!-- containers\application\AppParentDocument.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.*"
    width="100%" height="100%">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <!-- Include the AccChildObject.mxml component. -->
    <MyComps:AccChildObject/>
</s:Application>
```

In this example, the application is the parent document of the AccChildObject.mxml component. The following code from the AccChildObject.mxml component uses the `parentDocument` property to define an Accordion container that is slightly smaller than the Application container:

```
<?xml version="1.0"?>
<!-- containers\application\myComponents\AccChildObject.mxml -->
<mx:Accordion xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="{parentDocument.width*.80}"
    height="{parentDocument.height*.50}">

    <mx:HBox label="Child HBox"/>

</mx:Accordion>
```

You use the `parentDocument` property in MXML scripts to go up a level in the chain of parent documents. You can use the `parentDocument` to walk this chain by using multiple `parentDocument` properties, as the following example shows:

```
parentDocument.parentDocument.doSomething();
```

The `parentDocument` property of the application object is a reference to the application.

The `parentDocument` is typed as Object so that you can access properties and methods on ancestor document objects without casting.

Every UIComponent class has an `isDocument` property that is set to `true` if that UIComponent class is a document object, and `false` if it is not.

If a UIComponent class is a document object, it has a `documentDescriptor` property. This is a reference to the descriptor at the top of the generated descriptor tree in the generated document class.

For example, suppose that AddressForm.mxml component creates a subclass of the Form container to define an address form, and the MyApp.mxml component creates two instances of it: `<AddressForm id="shipping">` and `<AddressForm id="billing">`.

In this example, the shipping object is a document object. Its `documentDescriptor` property corresponds to the `<mx:Form>` tag at the top of the AddressForm.mxml file (the definition of the component), while its descriptor corresponds to the `<AddressForm id="shipping">` tag in MyApp.mxml file (an instance of the component).

Walking the document chain by using the `parentDocument` property is similar to walking the application chain by using the `parentApplication` property.

Using the parentApplication property

Applications can load other applications; therefore, you can have a hierarchy of applications, similar to the hierarchy of documents within each application. Every UIComponent class has a `parentApplication` read-only property that references the application object in which the object exists. The `parentApplication` property of an application object is never itself; it is either the application object into which it was loaded, or it is `null` (for the application object).

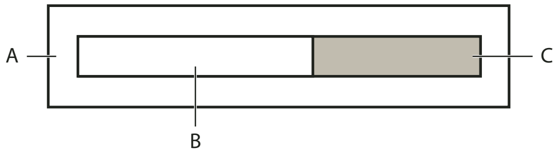
Walking the application chain by using the `parentApplication` property is similar to walking the document chain by using the `parentDocument` property.

Showing the download progress of an application

The application containers support an application preloader that uses a download progress bar to show the download and initialization progress of an application SWF file. By default, the application preloader is enabled. The preloader tracks how many bytes have been downloaded and continually updates the progress bar.

The download progress bar displays information about two different phases of the application: the download phase and the initialization phase. The application container's `creationComplete` event dismisses the preloader.

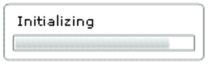
Flex includes two download progress bar classes: the SparkDownloadProgressBar (Spark) and the DownloadProgressBar (MX). The following image shows the Spark download progress bar:



A. Spark download progress bar. B. Initialization progress track. C. Download progress track.

When the SWF file is downloading, download progress track expands. When the application is initializing, the initilization progress track expands. The SparkDownloadProgressBar class does not contain text.

The following example shows the MX download progress bar during the initialization phase:



The MX download progress bar supports text. The bar displays the text "Downloading app" during the SWF download. It displays the text "Initializing app" during application initialization.

The download progress bar is not displayed if the SWF file is on your local host or if it is already cached. If the SWF file is not on your local host and is not cached, the progress bar is displayed if less than half of the application is downloaded after 700 milliseconds of downloading.

Adding a splash screen

As an alternative to the preloader, you can display a splash screen instead. The splash screen appears during the time of application startup. For information on using a splash screen, see Add a splash screen to an application.

Setting the download progress bar class

By default, an Application container uses the SparkDownloadProgressBar class. To configure the Application container to use the MX class, DownloadProgressBar, use the `preloader` property as the following example shows:

```
<s:Application ... preloader="mx.preloaders.DownloadProgressBar">
```

Disabling the download progress bar

To disable the download progress bar, set the `usePreloader` property of the Application container to `false`, as the following example shows:

```
<s:Application ... usePreloader="false">
```

Creating a custom progress bar

To create a custom download progress bar, you can create a subclass of the SparkDownloadProgressBar or DownloadProgressBar class, or create a subclass of the flash.display.Sprite class that implements the mx.preloaders.IPreloaderDisplay interface.

You can implement a download progress bar component as a SWC component or an ActionScript component. A custom download progress bar component that extends the Sprite class should not use any of the standard Flex components because it would load too slowly to be effective. Do not implement a download progress bar as an MXML component because it also would load too slowly.

To use a custom download progress bar class, you set the `preloader` property of the application container to the path of a SWC component class or ActionScript component class. A SWC component must be in the same directory as the MXML file or in a directory on the classpath of your application. An ActionScript component can be in one of those directories or in a subdirectory of one of those directories. When a class is in a subdirectory, you specify the subdirectory location as the package name in the `preloader` value; otherwise, you specify the class name.

The code in the following example specifies a custom download progress bar called CustomBar that is located in the myComponents/mybars directory below the application's root directory:

```
<s:Application ... preloader="myComponents.mybars.CustomBar">
```

Download progress bar events

The operation of the download progress bar is defined by a set of events. These events are dispatched by the Preloader class. The SparkDownloadProgressBar and DownloadProgressBar classes defines an event listener for all these events.

Within your custom class, you can optionally override the default behavior of the event listener. If you create a custom download progress bar as a subclass of the Sprite class, define an event listener for each of these events.

The following table describes the download progress bar events:

Event	Description
`ProgressEvent.PROGRESS`	Dispatched when the application SWF file is being downloaded. The first `PROGRESS` event signifies the beginning of the download process.
`Event.COMPLETE`	Dispatched when the SWF file has finished downloading. Either zero or one `COMPLETE` event is dispatched.
`FlexEvent.INIT_COMPLETE`	Dispatched when the application finishes initialization. This event is always dispatched once, and is the last event that the Preloader dispatches. The download progress bar must dispatch a `COMPLETE` event after it has received an `INIT_COMPLETE` event. The `COMPLETE` event informs the Preloader that the download progress bar has completed all operations and can be dismissed. The download progress bar can perform additional tasks, such as playing an animation, after receiving an `INIT_COMPLETE` event, and before dispatching the COMPLETE event. Dispatching the COMPLETE event should be the last action of the download progress bar.
`FlexEvent.INIT_PROGRESS`	Dispatched when the application completes an initialization phase, as defined by calls to the `measure()`, `commitProperties()`, or `updateDisplayList()` methods. This event describes the progress of the application in the initialization phase.

Event	Description
`RSLEvent.RSL_ERROR`	Dispatched when a Runtime Shared Library (RSL) fails to load.
`RSLEvent.RSL_LOADED`	Dispatched when an RSL finishes loading. The total bytes and total loaded bytes are included in the event object. This event is dispatched for every RSL that is successfully loaded.
`RSLEvent.RSL_PROGRESS`	Dispatched when an RSL is being downloaded. The first progress event signifies the beginning of the RSL download. The event object for this event is of type RSLEvent.

Creating a simple download progress bar class

The easiest way to create your own download progress bar is to create a subclass of the SparkDownloadProgressBar or DownloadProgressBar class, and then modify it for your application requirements.

The following example shows a custom download progress bas based on the SparkDownloadProgressbar class. This class overrides the getter methods for the `SparkDownloadProgressbar.backgroundImage` and `SparkDownloadProgressbar.backgroundSize` properties to show an image during application download and inititialization:

```
package myComponents
{
    import mx.preloaders.*;
    import flash.events.ProgressEvent;
    public class SparkDownloadProgressBarSubClassMin extends SparkDownloadProgressBar
    {
        public function SparkDownloadProgressBarSubClassMin() {
            super();
        }

        // Embed the background image.
        [Embed(source="logo.jpg")]
        [Bindable]
        public var imgCls:Class;
        // Override to set a background image.
        override public function get backgroundImage():Object{
            return imgCls;
        }

        // Override to set the size of the background image to 100%.
        override public function get backgroundSize():String{
            return "100%";
        }

        // Override to return true so progress bar appears
        // during initialization.
        override protected function showDisplayForInit(elapsedTime:int,
            count:int):Boolean {
                return true;
        }
        // Override to return true so progress bar appears during download.
        override protected function showDisplayForDownloading(
            elapsedTime:int, event:ProgressEvent):Boolean {
                return true;
        }
    }
}
```

The following application uses this custom class:

```
<?xml version="1.0"?>
<!-- containers\application\SparkMainDPBMin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    preloader="myComponents.SparkDownloadProgressBarSubClassMin">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Add a couple of controls that don't do anything. -->
    <s:Button label="Click Me"/>
    <s:TextInput text="This is a TextInput control."/>
</s:Application>
```

The Spark download progress bar does not support text, but the MX one does. The next example creates a subclass of the MX DownloadProgressBar class to define custom strings for the download progress bar, and set the minimum time that it appears, as the following example shows:

```
package myComponents
{
    import mx.preloaders.*;
    import flash.events.ProgressEvent;
    public class DownloadProgressBarSubClassMin extends DownloadProgressBar
    {
        public function DownloadProgressBarSubClassMin()
        {
            super();
            // Set the download label.
            downloadingLabel="Downloading app..."
            // Set the initialization label.
            initializingLabel="Initializing app..."
            // Set the minimum display time to 2 seconds.
            MINIMUM_DISPLAY_TIME=2000;
        }

        // Override to return true so progress bar appears
        // during initialization.
        override protected function showDisplayForInit(elapsedTime:int,
            count:int):Boolean {
                return true;
        }
        // Override to return true so progress bar appears during download.
        override protected function showDisplayForDownloading(
            elapsedTime:int, event:ProgressEvent):Boolean {
                return true;
        }
    }
}
```

You can use your custom class in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\MainDPBMin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    preloader="myComponents.DownloadProgressBarSubClassMin">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <!-- Add a couple of controls that don't do anything. -->
    <s:Button label="Click Me"/>
    <s:TextInput text="This is a TextInput control."/>
</s:Application>
```

Creating an advanced subclass of the DownloadProgressBar class

In the following example, you create a subclass of the DownloadProgressBar class to display text messages that describe the status of the downloading and initialization of the application. This example defines event listeners for the events dispatched by the download progress bar to write the messages to flash.text.TextField objects.

```
package myComponents
{
    import flash.display.*;
    import flash.text.*;
    import flash.utils.*;
    import flash.events.*;
    import mx.preloaders.*;
    import mx.events.*;
    public class MyDownloadProgressBar extends DownloadProgressBar
    {
        // Define a TextField control for text messages
        // describing the download progress of the application.
        private var progressText:TextField;

        // Define a TextField control for the final text message.
        // after the application initializes.
        private var msgText:TextField;

        public function MyDownloadProgressBar()
        {
            super();
            // Configure the TextField for progress messages.
            progressText = new TextField();
            progressText.x = 10;
            progressText.y = 90;
            progressText.width = 400;
            progressText.height = 400;

            addChild(progressText);

            // Configure the TextField for the final message.
            msgText = new TextField();
            msgText.x = 10;
            msgText.y = 10;
            msgText.width = 400;
            msgText.height = 75;

            addChild(msgText);
        }

        // Define the event listeners for the preloader events.
        override public function set preloader(preloader:Sprite):void {
            // Listen for the relevant events
            preloader.addEventListener(
                ProgressEvent.PROGRESS, myHandleProgress);
            preloader.addEventListener(
                Event.COMPLETE, myHandleComplete);
            preloader.addEventListener(
                FlexEvent.INIT_PROGRESS, myHandleInitProgress);
            preloader.addEventListener(
                FlexEvent.INIT_COMPLETE, myHandleInitEnd);
        }

        // Event listeners for the ProgressEvent.PROGRESS event.
        private function myHandleProgress(event:ProgressEvent):void {
            progressText.appendText("\n" + "Progress l: " +
                event.bytesLoaded + " t: " + event.bytesTotal);
        }

        // Event listeners for the Event.COMPLETE event.
        private function myHandleComplete(event:Event):void {
            progressText.appendText("\n" + "Completed");
        }

        // Event listeners for the FlexEvent.INIT_PROGRESS event.
        private function myHandleInitProgress(event:Event):void {
            progressText.appendText("\n" + "App Init Start");
        }

        // Event listeners for the FlexEvent.INIT_COMPLETE event.
        private function myHandleInitEnd(event:Event):void {
            msgText.appendText("\n" + "App Init End");

            var timer:Timer = new Timer(2000,1);
            timer.addEventListener(TimerEvent.TIMER, dispatchComplete);
            timer.start();
        }

        // Event listener for the Timer to pause long enough to
        // read the text in the download progress bar.
        private function dispatchComplete(event:TimerEvent):void {
            dispatchEvent(new Event(Event.COMPLETE));
        }
    }
}
```

You can use your custom class in a application, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\MainDPB.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    preloader="myComponents.MyDownloadProgressBar">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Add a couple of controls that don't do anything. -->
    <s:Button label="Click Me"/>
    <s:TextInput text="This is a TextInput control."/>
</s:Application>
```

Creating a subclass of Sprite as a download progress bar

You can define a custom download progress bar as a subclass of the Sprite class. By implementing your download progress bar as a subclass of Sprite, you can create a custom look and feel to it, rather than overriding the behavior built into the DownloadProgressBar class.

One common use for this type of download progress bar is to have it display a SWF file during application initialization. For example, you could display a SWF file that shows a running clock, or other type of image.

The following example displays a SWF file as the download progress bar. This class must implement the IPreloaderDisplay interface.

```
package myComponents
{
    import flash.display.*;
    import flash.utils.*;
    import flash.events.*;
    import flash.net.*;
    import mx.preloaders.*;
    import mx.events.*;

    public class MyDownloadProgressBarSWF extends Sprite
        implements IPreloaderDisplay
    {
        // Define a Loader control to load the SWF file.
        private var dpbImageControl:flash.display.Loader;

        public function MyDownloadProgressBarSWF() {
            super();
        }

        // Specify the event listeners.
        public function set preloader(preloader:Sprite):void {
            // Listen for the relevant events
            preloader.addEventListener(
                ProgressEvent.PROGRESS, handleProgress);
            preloader.addEventListener(
                Event.COMPLETE, handleComplete);

            preloader.addEventListener(
                FlexEvent.INIT_PROGRESS, handleInitProgress);
            preloader.addEventListener(
                FlexEvent.INIT_COMPLETE, handleInitComplete);
        }

        // Initialize the Loader control in the override
        // of IPreloaderDisplay.initialize().
        public function initialize():void {
            dpbImageControl = new flash.display.Loader();
            dpbImageControl.contentLoaderInfo.addEventListener(
                Event.COMPLETE, loader_completeHandler);
            dpbImageControl.load(new URLRequest("assets/dpbSWF.swf"));
        }
        // After the SWF file loads, set the size of the Loader control.
        private function loader_completeHandler(event:Event):void
        {
            addChild(dpbImageControl);
            dpbImageControl.width = 50;
            dpbImageControl.height= 50;
            dpbImageControl.x = 100;
            dpbImageControl.y = 100;
        }

        // Define empty event listeners.
        private function handleProgress(event:ProgressEvent):void {
        }

        private function handleComplete(event:Event):void {
        }
        private function handleInitProgress(event:Event):void {
        }

        private function handleInitComplete(event:Event):void {
            var timer:Timer = new Timer(2000,1);
            timer.addEventListener(TimerEvent.TIMER, dispatchComplete);
            timer.start();
        }

        private function dispatchComplete(event:TimerEvent):void {
            dispatchEvent(new Event(Event.COMPLETE));
        }
        // Implement IPreloaderDisplay interface

        public function get backgroundColor():uint {
            return 0;
        }

        public function set backgroundColor(value:uint):void {
        }

        public function get backgroundAlpha():Number {
            return 0;
        }

        public function set backgroundAlpha(value:Number):void {
        }

        public function get backgroundImage():Object {
            return undefined;
        }

        public function set backgroundImage(value:Object):void {
        }

        public function get backgroundSize():String {
            return "";
        }

        public function set backgroundSize(value:String):void {
        }

        public function get stageWidth():Number {
            return 200;
        }

        public function set stageWidth(value:Number):void {
        }

        public function get stageHeight():Number {
            return 200;
        }

        public function set stageHeight(value:Number):void {
        }
    }
}
```

Spark containers

Spark containers provide a hierarchical structure to arrange and configure their children.

Flex also provides a set of MX layout and MX navigator containers. Adobe recommends that you use the Spark containers when possible. For more information on the MX layout and MX navigator containers, see “MX layout containers” on page 574 and “MX navigator containers” on page 628.

For an introduction to containers, including Spark containers, see “Introduction to containers” on page 326.

About Spark containers

Spark includes the following containers:

- Group (including HGroup, VGroup, and TileGroup) and DataGroup
- SkinnableContainer, SkinnableDataContainer, Panel, TitleWindow, NavigatorContent, BorderContainer, and Application

  ***Note:** The Panel and NavigatorContent classes are subclasses of the SkinnableContainer class. The information below for SkinnableContainer applies also to the Panel and NavigatorContent classes.*

For more information on the Application container, see “Application containers” on page 393.

Interchangeable layouts

Most Spark containers support interchangeable layouts. That means you can set the layout of a container to any of the supported layout types, such as basic, horizontal, vertical, or tiled layout. You can also define a custom layout.

***Note:** Some Spark containers, such as HGroup, VGroup, and TileGroup, have a predefined layout that you cannot change.*

Skinning

To improve performance and minimize application size, some Spark containers do not support skinning. Use the Group and DataGroup containers to manage child layout. Use SkinnableContainer and SkinnableDataContainer to manage child layout and to support custom skins.

Visual child components

The Group and SkinnableContainer classes can take any visual components as children. Visual components implement the IVisualElement interface, and include subclasses of the UIComponent class and the GraphicElement class.

The UIComponent class is the base class of all Flex components. Therefore, you can use any Flex component as a child of the Group and SkinnableContainer class.

The GraphicElement class is the base class for the Flex drawing classes, such as the Ellipse, Line, and Rect classes. Therefore, you can use subclass of the GraphicElement class as a child of the Group and SkinnableContainer class.

The DataGroup and SkinnableDataContainer classes take as children visual components that implement the IVisualElement interface and are subclasses of DisplayObject. This includes subclasses of the UIComponent class.

Data items

However, the DataGroup and SkinnableDataContainer containers are optimized to hold data items. Data items can be simple data items such as String and Number objects, and more complicated data items such as Object and XMLNode objects. Therefore, while these containers can hold visual children, use Group and SkinnableContainer for children that are visual components.

Main characteristics of Spark containers

The following table lists the main characteristics of the Spark containers:

Container	Children	Skinnable	Scrollable	Creation policy	Primary use
Group (including HGroup, VGroup, and TileGroup)	IVisualElement	No	As a child of Scroller	All	Lay out visual children.
DataGroup	Data Item IVisualElement and DisplayObject	No	As a child of Scroller	All	Render and lay out data items.
SkinnableContainer	IVisualElement	Yes	By skinning	Selectable	Lay out visual children in a skinnable container.
SkinnablePopUpContainer	IVisualElement	Yes	By skinning	Selectable	Lay out visual children in a skinnable container opened as a pop up window.
SkinnableDataContainer	Data item IVisualElement and DisplayObject	Yes	By skinning	Selectable	Render and lay out data items in a skinnable container.
BorderContainer	IVisualElement	Yes	No	Selectable	Lay out visual children in a basic container that includes a border.
Form	IVisualElement	Yes	Yes	Selectable	Lay out FormItem and FormHeading children similar to an HTML form.
NavigatorContent	IVisualElement	Yes	By skinning	Inherited from parent container	Subclass of SkinnableContainer that can be used as the child of an MX navigator container.

Container	Children	Skinnable	Scrollable	Creation policy	Primary use
Panel	IVisualElement	Yes	By skinning	Selectable	Subclass of SkinnableContainer that adds a title bar and other visual elements to the container.
TitleWindow	IVisualElement	Yes	By skinning	Selectable	Subclass of Panel that is optimized for use as a pop-up window.

For information on skinning, see "Spark Skinning" on page 1602.

Adobe Evangelist Renaun Erickson explores the Flex 4 container classes in Migrating to Flex 4: Containers.

Creation policy

Container creation policy determines how and when children of containers are created. For information on creation policy, see "About the creation policy" on page 327.

About container children

All Spark containers can take as children visual components that implement the IVisualElement interface. All Spark and MX components implement the IVisualElement interface and can therefore be used as container children.

The following example uses the Spark SkinnableContainer to hold Spark Button components:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:SkinnableContainer>
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:SkinnableContainer>
</s:Application>
```

The Spark Group, SkinnableContainer, and subclasses can take as children any subclass of the GraphicElement class. This lets you add graphical elements as children of the container. The following example shows a SkinnableContainer with a Line component between two Button components:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerGraphic.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableContainer>
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Line
            xFrom="0" xTo="100">
            <s:stroke>
                <s:LinearGradientStroke weight="2"/>
            </s:stroke>
        </s:Line>
        <s:Button label="Button 2"/>
    </s:SkinnableContainer>
</s:Application>
```

The DataGroup and SkinnableDataContainer classes are designed primarily to display data items as children. The following example shows a SkinnableDataContainer displaying an Array of Strings as children:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableDataContainer itemRenderer="spark.skins.spark.DefaultItemRenderer">
        <mx:ArrayList>
            <fx:String>Dave Jones</fx:String>
            <fx:String>Mary Davis</fx:String>
            <fx:String>Debbie Cooper</fx:String>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

About Spark layouts

All Spark containers define a default layout, but let you switch the layout to suit your application requirements. To switch layout, assign a layout class to the `layout` property of the container.

Flex ships with several layout classes that you can use with the Spark containers. Additionally, you can define custom layout classes. The layout classes are defined in the spark.layouts package, and include the following classes:

- BasicLayout Uses absolute positioning. You explicitly position all container children, or use constraints to position them.
- HorizontalLayout Lays out children in a single horizontal row. The height of the row is fixed to the same height for all children and is typically the height of the tallest child. The width of each child is either fixed to the same value for all children, or each child can calculate its own width. By default, each child calculates its own width.
- TileLayout Lays out children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The `orientation` property determines the layout direction. The valid values for the orientation property are `columns` for a column layout and `rows` (default) for a row layout.

  All cells of the tile layout have the same size, which is the height of the tallest child and the width of the widest child.

- VerticalLayout Lays out children in a single vertical column. The width of the column is fixed to the same width for all children and is typically the width of the widest child. The height of each child is either fixed to the same value for all children, or each child can calculate its own height. By default, each child calculates its own height.

Blogger Xavi Beumala blogged about a custom layout: a 3D accordion.

Blogger Brian Telintelo blogged about Flex: Spark TileLayout Finally Pays off for Multi-Screen Apps.

Evtim Georgiev on the Flex SDK team has several blog entries on Spark layouts, including creating layout animations and extending HorizontalLayout.

Set the layout of a Spark container

By default, the Group container uses the BasicLayout class. The following example uses the `layout` property of the container to set its layout to the HorizontalLayout class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerHorizontal.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:Group>
</s:Application>
```

To simplify the use of layouts with the Group container, Flex defines subclasses of Group: HGroup with a horizontal layout, VGroup with a vertical layout, and TileGroup with a tile layout. Therefore, you can rewrite the previous example as shown below, replacing the HorizontalLayout definition inside Group with an HGroup container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkHGroupContainer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:HGroup>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:HGroup>
</s:Application>
```

Set the padding and gap of a layout

Most Spark layout classes support padding and gap properties. The padding properties define the space between the container boundaries and the children. The gap properties define the space between the children, either horizontally or vertically. The BasicLayout class does not support padding properties because this layout requires that you explicitly position each child.

The following example sets the padding to 10 pixels, and the gap to 5 pixels, for the children of a Panel container that uses the HorizontalLayout class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkPanelPadding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Panel>
        <s:layout>
            <s:HorizontalLayout
                paddingLeft="10" paddingRight="10"
                paddingTop="10" paddingBottom="10"
                gap="5"/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:Panel>
</s:Application>
```

For all Group containers, you set the padding with the `paddingLeft`, `paddingTop`, `paddingRight`, and `paddingBottom` properties. For VGroup and HGroup containers, you set the `gap` property. For the TileGroup container, you set the gap with the `horizontalGap` and `verticalGap` properties.

Set the alignment of a layout

The layout classes provide the `horizontalAlign` and `verticalAlign` properties for aligning the children of a container. For example, you can use these properties to align children to the top of a container using HorizontalLayout, or to the left side of a container using VerticalLayout.

horizontalAlign values

The `horizontalAlign` property takes the following values:

- `left` Align children to the left side of the container. This is the default value for HorizontalLayout.
- `right` Align children to the right side of the container.
- `center` Align the children to the horizontal center of the container.
- `justify` Set the width of all children to be the same width as the container.
- `contentJustify` Set the width of all children to be the *content width* of the container. The content width of the container is the width of the largest child. If all children are smaller than the width of the container, then set the width of all the children to the width of the container.

  Not supported by TileLayout. Use `columnAlign` and `rowAlign` instead.

verticalAlign values

The `verticalAlign` property takes the following values:

- `top` Align children to the top of the container. This is the default value for VerticalLayout.
- `bottom` Align children to the bottom of the container.
- `middle` Align children to the vertical middle of the container.
- `baseline` Align the elements such that their text is aligned to the maximum of the elements' text ascent.
- `justify` Set the height of all children to be the same height as the container.

- `contentJustify` Set the height of all children to be the *content height* of the container. The content height of the container is the height of the largest child. If all children are smaller than the height of the container, then set the height of all the children to the height of the container.

  Not supported by TileLayout. Use `columnAlign` and `rowAlign` instead.

The following example overrides the default vertical alignment of top for the HorizontalLayout to align the children of a Group container to the bottom of the container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimpleAlignment.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:SkinnableContainer>
        <s:layout>
            <s:HorizontalLayout verticalAlign="bottom"/>
        </s:layout>
        <s:Button label="Button 1" fontSize="24"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3" fontSize="36"/>
        <s:Button label="Button 4"/>
    </s:SkinnableContainer>
</s:Application>
```

The following example overrides the default vertical alignment of top for the HorizontalLayout to use justify. Notice that all buttons have the same height:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimpleAlignment.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:SkinnableContainer>
        <s:layout>
            <s:HorizontalLayout verticalAlign="justify"/>
        </s:layout>
        <s:Button label="Button 1" fontSize="24"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3" fontSize="36"/>
        <s:Button label="Button 4"/>
    </s:SkinnableContainer>
</s:Application>
```

Set the row height or column width of a layout

You control the way the container lays out children with different sizes by using the `VerticalLayout.variableRowHeight` and `HorizontalLayout.variableColumnWidth` properties. By default, these properties are set to `true` which lets each child determine its height (VerticalLayout) or width (HorizontalLayout).

In the following example, the Group container holds four buttons. By default, buttons use a 12 point font size. However, two of the buttons in this example define a larger font size. Because the `variableRowHeight` property is set to `true` by default, the container sets the height of each button appropriately for its font size:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightTrue.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:layout>
            <s:VerticalLayout />
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3" fontSize="36"/>
        <s:Button label="Button 4" fontSize="24"/>
    </s:Group>
</s:Application>
```

Variable row and column sizes

If your container has many children, setting the `variableRowHeight` or `variableColumnWidth` properties to `true` can affect application performance. The reason is because the container calculates the size of every child as it appears on the screen.

Rather than having each child calculate its size, you can set the `variableRowHeight` or `variableColumnWidth` property to `false` so that each child has the same size.

***Note:** The following paragraphs describe setting the `variableRowHeight` property for the VerticalLayout class. This discussion is the same as setting the `variableColumnWidth` property for the HorizontalLayout class.*

If you set the `variableRowHeight` property to `false`, the VerticalLayout class uses the following procedure to determine the height of each child:

1. If specified, use the `VerticalLayout.rowHeight` property to specify an explicit height of all children. Make sure that the specified height is suitable for all children.
2. If specified, use the `VerticalLayout.typicalLayoutElement` property to define the height of all children. This property references a component that Flex uses to define the height of all container children.
3. Use the preferred height of the first container child as the height of all container children. This technique is useful if the children of the container are all similar.

In the following example, a group container uses the VerticalLayout class to lay out four Button controls. The `variableRowHeight` property is `false` so that every button has the same height:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightFalse.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:layout>
            <s:VerticalLayout variableRowHeight="false"/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3" fontSize="36"/>
        <s:Button label="Button 4" fontSize="24"/>
    </s:Group>
</s:Application>
```

Because you did not specify an explicit value for the `VerticalLayout.rowHeight` property or the `VerticalLayout.typicalLayoutElement` property, the VerticalLayout class uses the preferred height of the first button control as the height for all container children. However, because the third button and fourth button controls define a large font size, the text is truncated to the size of the button.

Alternatively, you can set the `rowHeight` property to a pixel value large enough for all the children, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightFalseRowHeight.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:layout>
            <s:VerticalLayout variableRowHeight="false"
                rowHeight="40"/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3" fontSize="36"/>
        <s:Button label="Button 4" fontSize="24"/>
    </s:Group>
</s:Application>
```

In this example, all buttons are 40 pixels tall.

The following example uses the `typicalLayoutElement` property to specify to use the third button to determine the height of all container children:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightFalseTypicalLE.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:layout>
            <s:VerticalLayout variableRowHeight="false"
                typicalLayoutElement="{b3}"/>
        </s:layout>
        <s:Button id="b1" label="Button 1"/>
        <s:Button id="b2" label="Button 2"/>
        <s:Button id="b3" label="Button 3" fontSize="36"/>
        <s:Button id="b4" label="Button 4" fontSize="24"/>
    </s:Group>
</s:Application>
```

You can use two common strategies for determining the typical item. One option is to use the largest item as the typical item. Another option is to calculate the average size of all items, and use the data item closest to that average size.

The Spark Group and Spark SkinnableContainer containers

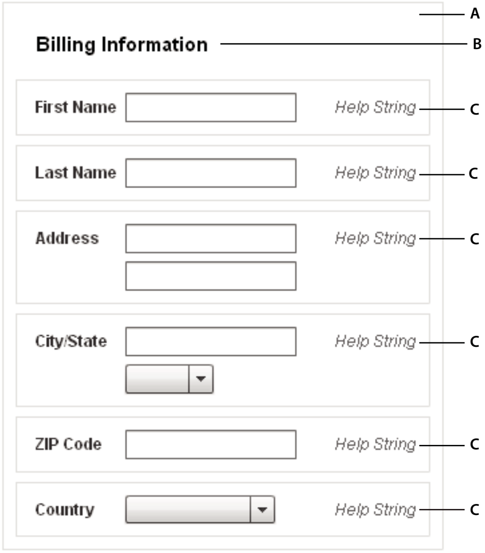
The Spark Group and Spark SkinnableContainer containers take as children any components that implement the IVisualElement interface. Use these containers when you want to manage visual children, both visual components and graphical components.

The main differences between the Group and SkinnableContainer containers are:

- SkinnableContainer can be skinned. The Group container is designed for simplicity and minimal overhead, and cannot be skinned.
- Group can be a child of the Scroller control to support scroll bars. Create a skin for the SkinnableContainer to add scroll bars.

One of the uses of the Group container is to import graphic elements from Adobe design tools, such as Adobe Illustrator. For example, if you use a design tool to create graphics imported into Flex, the graphics are often represented using FXG statements in a Group container. For more information, see “FXG and MXML graphics” on page 1714.

The default layout class of the Group and SkinnableContainer container is BasicLayout. The following example shows a Group container with a horizontal layout and one with a vertical layout:



For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a Spark Group container

You use the `<s:Group>` tag to define a Group container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example shows a Group container with four Button controls as its children:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:Button label="Button 1"
            left="10" top="13" bottom="10"/>
        <s:Button label="Button 2"
            left="110" top="13" bottom="10"/>
        <s:Button label="Button 3"
            left="210" top="13" bottom="10"/>
        <s:Button label="Button 4"
            left="310" top="13" bottom="10" right="10"/>
    </s:Group>
</s:Application>
```

In this example, the Group container uses its default layout specified by the BasicLayout class, which means it uses absolute layout. The four button controls then use constraints to set their positions in the container. For more information on constraints, see “Using constraints to control component layout” on page 384.

You can add a graphic element to the container to define a background for the buttons, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerRect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:Rect x="0" y="0"
            radiusX="4" radiusY="4"
            height="100%" width="100%">
            <s:stroke>
                <s:LinearGradientStroke weight="1" scaleMode="normal"/>
            </s:stroke>
            <s:fill>
                <s:LinearGradient>
                    <s:entries>
                        <mx:GradientEntry color="0x999999"/>
                    </s:entries>
                </s:LinearGradient>
            </s:fill>
        </s:Rect>
        <s:Button label="Button 1"
            left="10" top="13" bottom="10"/>
        <s:Button label="Button 2"
            left="110" top="13" bottom="10"/>
        <s:Button label="Button 3"
            left="210" top="13" bottom="10"/>
        <s:Button label="Button 4"
            left="310" top="13" right="10" bottom="10"/>
    </s:Group>
</s:Application>
```

In this example, you add an instance of the Rect class, a subclass of GraphicElement, that defines a gray background and one pixel border around the container. In this example the Rect is located a coordinates 0,0 in the Group container, and sized to fill the entire container.

Creating a Spark SkinnableContainer container

You use the `<s:SkinnableContainer>` tag to define a SkinnableContainer container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example shows a SkinnableContainer container with four Button controls as its children:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:SkinnableContainer>
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:SkinnableContainer>
</s:Application>
```

The default layout class of the SkinnableContainer class is BasicLayout. In this example, the SkinnableContainer uses the HorizontalLayout class to arrange the buttons in a single row.

If the SkinnableContainer uses BasicLayout, you can use a Rect component as a child of the container to add a background color and border. For an example, see “Creating a Spark Group container” on page 426.

However, the SkinnableContainer class lets you apply a skin to define the visual characteristics of the container. For example, the following skin defines a gray background and a one pixel border for the container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\mySkins\MyBorderSkin.mxml -->
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <!-- Define the skin states. -->
    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>

    <!-- Define a Rect to fill the area of the skin. -->
    <s:Rect x="0" y="0"
        radiusX="4" radiusY="4"
        height="100%" width="100%">
        <s:stroke>
            <s:LinearGradientStroke weight="1"/>
        </s:stroke>
        <s:fill>
            <s:LinearGradient>
                <s:entries>
                   <mx:GradientEntry color="0x999999"/>
                </s:entries>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>
    <!-- Define the content area of the container. -->
    <s:Group id="contentGroup"
        left="5" right="5" top="5" bottom="5">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
</s:Skin>
```

All SkinnableContainer skins must implement the view states defined by the SkinnableContainer. Because the SkinnableContainer class supports the normal and disabled view states, the skin must also support them.

The Rect component adds the border and gray background to the skin.

All the container's children are added to the `contentGroup` skin part of the skin. In this example, the `contentGroup` container is a Group container with a vertical layout. Setting the `layout` property of the SkinnableContainer overrides the layout specified in the skin.

The advantage to defining a skin for the SkinnableContainer, rather than adding the visual elements in the SkinnableContainer definition, is that the skin is reusable. For example, you typically define a consistent look for all SkinnableContainer containers in an application. By encapsulating that look in a reusable skin class, you can apply it to all containers in your application.

Use the `skinClass` property to apply the skin to the SkinnableContainer, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:SkinnableContainer
        skinClass="mySkins.MyBorderSkin">
        <s:layout>
            <s:HorizontalLayout gap="10"/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:SkinnableContainer>
</s:Application>
```

For more information on skinning, see “Spark Skinning” on page 1602.

The Spark BorderContainer container

To change the visible appearance of a SkinnableContainer, create a custom skin. However, to make it simple to add a border or change the background of a container, Flex defines the Spark BorderContainer container as a subclass of the SkinnableContainer. The BorderContainer container provides a set of CSS styles and class properties to control the border and background of the container.

***Note:** Because you use CSS styles and class properties to control the appearance of the BorderContainer container, you typically do not create a custom skin for it. If you do create a custom skin, your skin class should apply any styles to control the appearance of the container.*

The following example uses the `backgroundColor`, `borderStyle`, `borderWeight`, and `cornerRadius` styles of the BorderContainer container to control the appearance of the BorderContainer container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkBorderSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:BorderContainer
        backgroundColor="red" cornerRadius="10"
        borderStyle="inset" borderWeight="4" >
        <s:layout>
            <s:HorizontalLayout
                paddingLeft="5" paddingRight="5"
                paddingTop="5" paddingBottom="5"/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:BorderContainer>
</s:Application>
```

Use the `backgroundImage` style to specify an image as the background of the container, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkBorderImage.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            [Embed(source="/assets/logo.jpg")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </fx:Script>

    <s:BorderContainer
        backgroundImage="{imgCls}"
        borderStyle="inset" borderWeight="2"
        width="150" height="180">
    </s:BorderContainer>
</s:Application>
```

The BorderContainer container defines two properties that also let you control the appearance of the container:

- The `backgroundFill` property, of type IFill, defines the fill of the container. If you set the `backgroundFill` property, then the container ignores the `backgroundAlpha`, `backgroundColor`, `backgroundImage`, and `backgroundImageResizeMode` styles.
- The `borderStroke` property, of type IStroke, defines the stroke of the border. If you set the `borderStroke` property, then the container ignores the `borderAlpha`, `borderColor`, `borderStyle`, `borderVisible`, and `borderWeight` styles.

The following example sets the `backgroundFill` and `borderStroke` properties:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkBorderFillStroke.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:BorderContainer cornerRadius="10">
        <s:layout>
            <s:HorizontalLayout
                paddingLeft="5" paddingRight="5"
                paddingTop="5" paddingBottom="5"/>
        </s:layout>
        <s:backgroundFill>
            <s:SolidColor
                color="red"
                alpha="100"/>
        </s:backgroundFill>
        <s:borderStroke>
            <mx:SolidColorStroke
                color="black"
                weight="3"/>
        </s:borderStroke>

        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:BorderContainer>
</s:Application>
```

Because the BorderContainer container does not implement the IViewport interface, it does not directly support scroll bars. However, you can wrap the Scroller control inside the BorderContainer container to add scroll bars, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkScrollBorder.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
    <s:BorderContainer width="100" height="100"
        borderWeight="3" borderStyle="solid">
        <s:Scroller width="100%" height="100%">
            <s:Group
                horizontalScrollPosition="50"
                verticalScrollPosition="50">
                <s:Image width="300" height="400"
                    source="@Embed(source='/assets/logo.jpg')"/>
            </s:Group>
        </s:Scroller>
    </s:BorderContainer>
</s:Application>
```

The Spark NavigatorContent container

The MX navigator containers include the Accordion, TabNavigator, and ViewStack containers. Navigator containers let users switch between multiple children, where the children are other containers. The child containers of a navigator can be any MX container and the Spark NavigatorContent container.

***Note:** You cannot use Spark containers other than NavigatorContent as a child of an MX navigator container. To use any other Spark container in a navigator container, wrap it in an MX container or in the Spark NavigatorContent container.*

For more information on MX navigator containers, see "MX navigator containers" on page 628.

Do not use a NavigatorContent container outside an MX navigator container. A NavigatorContent must be the child of an MX navigator container, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkNavContent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:Accordion>
        <s:NavigatorContent label="Pane 1"
            width="100" height="100">
            <s:layout>
                <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
            </s:layout>
            <s:Label text="Text for Pane 1"/>
            <s:Button label="Button 1"/>
        </s:NavigatorContent>
        <s:NavigatorContent label="Pane 2"
            width="100" height="100">
            <s:layout>
                <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
            </s:layout>
            <s:Label text="Text for Pane 2"/>
            <s:Button label="Button 2"/>
        </s:NavigatorContent>
        <s:NavigatorContent label="Pane 3"
            width="100" height="100">
            <s:layout>
                <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
            </s:layout>
            <s:Label text="Text for Pane 3"/>
            <s:Button label="Button 3"/>
        </s:NavigatorContent>
    </mx:Accordion>
</s:Application>
```

The value of the `label` property of each NavigatorContent child of the Accordion container defines the text that appears in the button area of the Accordion. You can use the `NavigatorContent.icon` property to specify an icon in the button area.

The creation policy of the NavigatorContent container is based on the creation policy of its parent navigator container, as the following table shows:

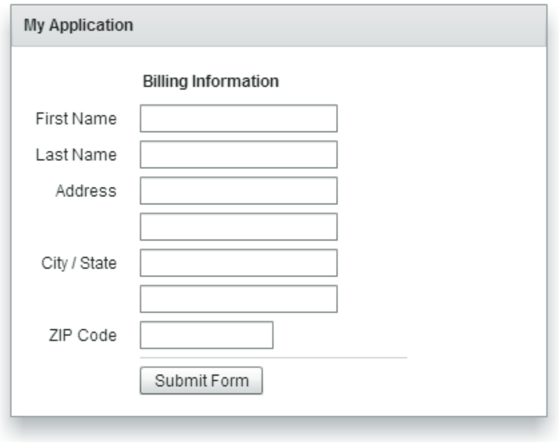
Creation policy of the parent navigator container	Creation policy of the NavigatorContent container
none	none
all	all
auto	none

For information on creation policy, see “About the creation policy” on page 327.

The Spark Panel container

A Panel container includes a title bar, a title, a border, and a content area for its children. Typically, you use Panel containers to wrap self-contained application modules. For example, you could define several Panel containers in your application where one Panel container holds a form, a second holds a shopping cart, and a third holds a catalog.

The default layout class of the Panel container is BasicLayout. The following example shows a Panel container with a vertical layout:



For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a Spark Panel container

You use the `<s:Panel>` tag to define a Panel container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example defines a Panel container that contains a form as the top-level container in your application. In this example, the Panel container provides you with a mechanism for including a title bar, as in a standard GUI window.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkPanelSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" >
    <s:Panel id="myPanel" title="My Application" x="20" y="20">
        <s:Form id="myForm" width="400">
            <s:FormHeading label="Billing Information"/>
            <s:FormItem label="First Name">
                <s:TextInput id="fname" width="100%"/>
            </s:FormItem>
            <s:FormItem label="Last Name">
                <s:TextInput id="lname" width="100%"/>
            </s:FormItem>
            <s:FormItem label="Address">
                <s:TextInput id="addr1" width="100%"/>
                <s:TextInput id="addr2" width="100%"/>
            </s:FormItem>
            <s:FormItem label="City">
                <s:TextInput id="city"/>
            </s:FormItem>

            <s:FormItem label="State">
                <s:TextInput id="state"/>
            </s:FormItem>

            <s:FormItem label="ZIP Code">
                <s:TextInput id="zip" width="100"/>
            </s:FormItem>
            <s:FormItem>
                <mx:HRule width="200" height="1"/>
                <s:Button label="Submit Form"/>
            </s:FormItem>
        </s:Form>
    </s:Panel>
</s:Application>
```

Adding a control bar to the Spark Panel container

A control bar contains a group of controls outside the main content area of a Spark Panel container. The control bar is always visible at the bottom of the Panel container. Therefore, if the Panel container uses scroll bars, the control bar is not scrolled along with the other container children.

Set the `controlBarVisible` property to `true` (the default value) to make the control bar visible. You use the `Panel.controlBarContent` property to define the controls that appear in the control bar area, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkPanelCB.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="750">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:XMLList id="menuXML">
            <fx:menuitem label="File">
                <fx:menuitem label="New" data="New"/>
                <fx:menuitem label="Open" data="Open"/>
                <fx:menuitem label="Save" data="Save"/>
                <fx:menuitem label="Exit" data="Exit"/>
            </fx:menuitem>
            <fx:menuitem label="Edit">
                <fx:menuitem label="Cut" data="Cut"/>
                <fx:menuitem label="Copy" data="Copy"/>
                <fx:menuitem label="Paste" data="Paste"/>
            </fx:menuitem>
            <fx:menuitem label="View"/>
        </fx:XMLList>

        <fx:Array id="cmbDP">
            <fx:String>Item 1</fx:String>
            <fx:String>Item 2</fx:String>
            <fx:String>Item 3</fx:String>
        </fx:Array>
    </fx:Declarations>
    <s:Panel title="Spark Panel">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:controlBarContent>
            <mx:MenuBar height="100%"
                dataProvider="{menuXML}"
                labelField="@label"
                showRoot="true"/>
            <mx:HBox paddingBottom="5"
                 paddingTop="5">
                <mx:ComboBox dataProvider="{cmbDP}"/>
                <mx:Spacer width="100%"/>
                <mx:TextInput id="myTI" text=""/>
                <mx:Button id="srch1"
                    label="Search"
                    click="Alert.show('Searching');"/>
            </mx:HBox>
        </s:controlBarContent>
        <s:Button label="Button"/>
        <s:TextArea width="300" height="200"/>
    </s:Panel>
</s:Application>
```

By default, the controls in the control bar area use horizontal layout. Use the `Panel.controlBarLayout` property to specify a different layout, as the following example shows:

```
<s:controlBarLayout>
    <s:HorizontalLayout paddingLeft="12" gap="5"/>
</s:controlBarLayout>
```

The location and appearance of the control bar area of the Panel container is determined by the spark.skins.spark.PanelSkin class, the skin class for the Panel container. By default, the PanelSkin class defines the control bar area to appear at the bottom of the content area of the Panel container with a grey background. Create a custom skin to change the default appearance of the control bar.

The Spark SkinnablePopUpContainer container

You use the SkinnablePopUpContainer container as a pop-up in your application. One typical use for a SkinnablePopUpContainer container is to open a simple window in an application, such as an alert window, to indicate that the user must perform some action.

The SkinnablePopUpContainer container can be modal or nonmodal. A modal container takes all keyboard and mouse input until it is closed. A nonmodal container lets other components accept input while the pop-up window is open.

Flex also defines the Spark TitleWindow container that you can use as a pop-up. A TitleWindow container consists of a title bar, a caption and status area in the title bar, a border, content area, and optional close button. Users can drag the TitleWindow container to move it around the application window. For more information, see “The Spark TitleWindow container” on page 443.

Unlike the TitleWindow container, the SkinnablePopUpContainer does not support dragging. It is a light-weight container that you use for simple pop-ups. You can define a custom skin class to control the visual aspects of the SkinnablePopUpContainer, such as the transitions and effects that play when the container opens and closes.

Creating a simple Spark SkinnablePopUpContainer container

The SkinnablePopUpContainer container appears as a pop-up window on top of its parent container. Therefore, you do not create a SkinnablePopUpContainer container as part of the normal MXML layout code of its parent container.

Instead, you can define the SkinnablePopUpContainer container as a custom MXML component in an MXML file. In the following example, you define it in the file MyAlertPopUp.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MyAlertPopUp.mxml -->
<s:SkinnablePopUpContainer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" width="400" height="300">
    <s:Panel title="My Alert Panel">
        <s:VGroup width="100%" horizontalAlign="center"
            paddingTop="20" gap="20">
                <s:Label text="Your custom alert box"/>
                <s:Button label="OK" click="close();"/>
        </s:VGroup>
    </s:Panel>
</s:SkinnablePopUpContainer>
```

The MyAlertPopUp.mxml component creates a SkinnablePopUpContainer that contains a Panel container. Notice that the Button control calls the `close()` method to close the pop-up.

At runtime, you create an instance of the SkinnablePopUpContainer in ActionScript, then call the following methods of the SkinnablePopUpContainer class to open and close it:

- `open()`

  Opens a SkinnablePopUpContainer container. The `open()` method takes two arguments: the first specifies the parent container of the SkinnablePopUpContainer container, and the second specifies if the container is modal.

  Opening the SkinnablePopUpContainer container dispatches the `open` event.

- `close()`

  Closes a SkinnablePopUpContainer container. The `close()` method takes two arguments: the first specifies whether the container passes back any data to the main application, and the second specifies an object containing the returned data.

  Closing the SkinnablePopUpContainer container dispatches the `close` event.

The following example shows the main application file that uses MyAlertPopUp.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkSkinnablePopUpContainerComponents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            // Import the MyAlertPopUp class.
            import myComponents.MyAlertPopUp;

            // Create an instance of MyAlertPopUp.
            public var alertDB:MyAlertPopUp = new MyAlertPopUp();
        ]]>
    </fx:Script>

    <!-- Open the MyAlertPopUp instance. -->
    <s:Button label="Open Alert" click="alertDB.open(this, false);"/>
</s:Application>
```

In this application, you first import the MyAlertPopUp component, then create an instance of it. The Button control calls the `open()` method to open the pop-up in response to a `click` event.

The first argument to the `open()` method specifies `this`. Therefore, the parent container of the SkinnablePopUpContainer container is the Application container. The second argument is `false` to specify a nonmodal window.

In this example, the application creates a single instance of the MyAlertPopUp component. It then reuses that instance every time the user selects the Button control. Therefore, the pop-up component stays in memory between uses.

If the pop-up component is large, or you want to reduce the memory use of the application, create a new instance of the component for each pop-up. The component is then destroyed when it closes. However, make sure to remove all references to the components, especially event handlers, or else the component is not destroyed.

Creating an inline Spark SkinnablePopUpContainer container

You do not have to define the SkinnablePopUpContainer container in a separate file. The following example uses the `<fx:Declaration>` tag to define an inline component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkSkinnablePopUpContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Declarations>
        <fx:Component className="MyAlert">
            <s:SkinnablePopUpContainer>
                <s:Panel title="My Alert Panel">
                    <s:VGroup width="100%" horizontalAlign="center"
                        paddingTop="20" gap="20">
                        <s:Label text="Your custom alert box"/>
                        <s:Button label="OK" click="close();"/>
                    </s:VGroup>
                </s:Panel>
            </s:SkinnablePopUpContainer>
        </fx:Component>
    </fx:Declarations>

    <s:Button label="Open Alert" click="(new MyAlert()).open(this, false);"/>
</s:Application>
```

The `<fx:Component>` tag defines a class named MyAlert. Notice that the `click` event handler for the Button control first creates a new instance of the MyAlert class, then opens the pop-up.

Alternatively, you could write the inline component so that it creates a single instance of the pop-up that is reused every time, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkSkinnablePopUpContainerInlineReuse.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Declarations>
        <s:SkinnablePopUpContainer id="alert">
            <s:Panel title="My Alert Panel">
                <s:VGroup width="100%" horizontalAlign="center"
                          paddingTop="20" gap="20">
                    <s:Label text="Your custom alert box"/>
                    <s:Button label="OK" click="alert.close();"/>
                </s:VGroup>
            </s:Panel>
        </s:SkinnablePopUpContainer>
    </fx:Declarations>

    <s:Button label="Open Alert" click="alert.open(this, false);"/>
</s:Application>
```

Controlling the pop-up window

You can use properties of the SkinnablePopUpContainer and the mx.managers.PopUpManager class to control the pop-up window. In the following example, you modify the width and height of the pop-up by setting properties of the SkinnablePopUpContainer. You then use the `PopUpManager.centerPopUp()` method to center it in its parent container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkSkinnablePopUpContainerMoveResize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.managers.PopUpManager;

            protected function button1_clickHandler(event:MouseEvent):void {

                // Create an instance of MyAlert.
                var alert:MyAlert = new MyAlert();
                alert.open(this, true);

                // Increase the width and height.
                alert.width += 100;
                alert.height += 100;

                // Center the pop-up in the parent container.
                PopUpManager.centerPopUp(alert);
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:Component className="MyAlert">
            <s:SkinnablePopUpContainer>
                <s:Panel title="My Alert Panel" width="100%" height="100%">
                    <s:VGroup width="100%" horizontalAlign="center"
                              paddingTop="20" gap="20">
                        <s:Label text="Your custom alert box"/>
                        <s:Button label="OK" click="close();"/>
                    </s:VGroup>
                </s:Panel>
            </s:SkinnablePopUpContainer>
        </fx:Component>
    </fx:Declarations>

    <s:Button label="Open Alert SMG" click="button1_clickHandler(event);"/>
</s:Application>
```

Passing data back from the Spark SkinnablePopUpContainer container

Use the `close()` method of the SkinnablePopUpContainer container to pass data back to the main application from the pop-up. The `close()` method has the following signature:

```
public function close(commit:Boolean = false, data:*):void
```

where:

- `commit` contains `true` if the returned data should be committed by the application.
- `data` specifies the returned data.

The `close()` method dispatches the `close` event. The event object associated with the `close` event is an object of type spark.events.PopUpEvent.

The PopUpEvent class defines two properties, `commit` and `data`, that contain the values of the corresponding arguments to the `close()` method. In the event handler of the `close` event, you use these properties to inspect any data returned from the pop-up.

The following example of the Spark SkinnablePopUpContainer contains a List control. If the user selects an item in the List, then the `close()` method passes that value back to the main application. If the user select the Cancel button, the `close()` method returns nothing:,

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MyAlertPopUpReturn.mxml -->
<s:SkinnablePopUpContainer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import spark.events.IndexChangeEvent;
            // Event handler for the change event of the List control.
            protected function list1_changeHandler(event:IndexChangeEvent):void {
                // Close the SkinnablePopUpContainer.
                // Set the commit argument to true to indicate that the
                // data argument contians a vaild value.
                close(true, list1.selectedItem);
            }
        ]]>
    </fx:Script>
    <s:Panel title="My Alert Panel">
        <s:VGroup width="100%" horizontalAlign="center"
                  paddingTop="20" gap="20">

            <s:List id="list1" width="100%" height="100%"
                    change="list1_changeHandler(event);">
                <s:layout>
                    <s:VerticalLayout/>
                </s:layout>
                <s:dataProvider>
                    <s:ArrayCollection source="{['Add', 'Delete', 'Post', 'Bookmark']}"/>
                </s:dataProvider>
            </s:List>

            <s:Button label="Cancel" click="close();"/>
        </s:VGroup>
    </s:Panel>

</s:SkinnablePopUpContainer>
```

Shown below is the main application that uses this component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkSkinnablePopUpContainerReturn.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            // Import the MyAlertPopUp class.
            import myComponents.MyAlertPopUpReturn;
            import spark.events.PopUpEvent;

            // Create and open the SkinnablePopUpContainer.
            protected function button1_clickHandler(event:MouseEvent):void {
                // Create an instance of MyAlertPopUp.
                var alertDB:MyAlertPopUpReturn = new MyAlertPopUpReturn();

                // Add an event handler for the close event to check for
                // any returned data.
                alertDB.addEventListener('close', alertDB_closeHandler);
                alertDB.open(this, true);
            }

            // Handle the close event from the SkinnablePopUpContainer.
            protected function alertDB_closeHandler(event:PopUpEvent):void {
                // If commit is false, do data is returned.
                if (!event.commit)
                    return;

                // Write the returned String to the TextArea control.
                myTA.text = event.data as String;
            }
        ]]>
    </fx:Script>

    <s:Button label="Open Alert" click="button1_clickHandler(event);"/>
    <s:TextArea id="myTA"/>
</s:Application>
```

This application uses the `click` event of the Button control to open the SkinnablePopUpContainer. The event handler for the `click` event creates an instance of the MyAlertPopUpReturn components, and then adds an event listener for the component's `close` event.

The event handler for the `close` event examines the `commit` property of the event object. If the `commit` property is `true`, the data returned from the MyAlertPopUpReturn component is written to the TextArea control.

The Spark TitleWindow container

A TitleWindow layout container is a Panel container that is optimized for use as a pop-up window. The container consists of a title bar, a caption and status area in the title bar, a border, and a content area for its children. Unlike the Panel container, it can display a close button. The TitleWindow container is designed to work as a pop-up window that users can drag around the screen the application window.

Modality

A pop-up TitleWindow container can be *modal*, which means that it takes all keyboard and mouse input until it is closed, or *nonmodal*, which means other windows can accept input while the pop-up window is still open.

Forms

One typical use for a TitleWindow container is to hold a form. When the user completes the form, you can close the TitleWindow container programmatically, or let the user request the application to close it by using the close icon (a box with an *x* inside it) in the upper-right corner of the window.

PopUpManager

Because you pop up a TitleWindow, you do not create it directly in MXML, as you do most controls. Instead you use the PopUpManager to open a TitleWindow container.

SkinnablePopUpContainer

Flex also defines the SkinnablePopUpContainer container. Unlike the TitleWindow container, the SkinnablePopUpContainer does not support dragging. It is a light-weight container that you use for simple pop ups. You can define a custom skin class to control the visual aspects of the SkinnablePopUpContainer, such as the transitions and effects that play when the container opens and closes. For more information, see "The Spark SkinnablePopUpContainer container" on page 437.

Using the PopUpManager to create a Spark TitleWindow container

To create and remove a pop-up TitleWindow container, you use methods of the PopUpManager. The PopUpManager is in the mx.managers package.

Creating and deleting Spark pop-up window

To create a pop-up window, use the PopUpManager `createPopUp()` method. The `createPopUp()` method has the following signature:

```
public static createPopUp(parent:DisplayObject, class:Class,
    modal:Boolean = false):IFlexDisplayObject
```

The method takes the following arguments.

Argument	Description
parent	A reference to a window to pop-up over.
class	A reference to the class of object you want to create. This is typically a custom MXML component that implements a Spark or MX TitleWindow container.
modal	(Optional) A Boolean value that indicates whether the window is modal, and takes all keyboard and mouse input until it is closed (`true`), or whether interaction is allowed with other controls while the window is displayed (`false`). The default value is `false`.

***Note:** Flex continues executing code in the parent even after you create a modal pop-up window.*

You use the `createPopUp()` method to create a pop-up window from a component definition. Typically, that means you have created an MXML component based on the TitleWindow container that defines your pop-up window. You can also create a pop-up window by passing an instance of a TitleWindow class or custom component to the PopUpManager `addPopUp()` method.

To remove a pop-up window, use the PopUpManager `removePopUp()` method. Pass the object created with the `createPopUp()` method or the `addPopUp()` method to the `removePopUp()` method.

Defining a custom Spark TitleWindow component

One of the most common ways of creating a TitleWindow container is to define it as a custom MXML component. Define the TitleWindow container, its event handlers, and all of its children in the custom component. Then, use the PopUpManager `createPopUp()` and `removePopUp()` methods to create and remove the TitleWindow container.

The following example code defines a custom MyLoginForm TitleWindow component that is used as a pop-up window:

```
<?xml version="1.0"?>
<!-- containers\spark\myComponents\MyLoginForm.mxml -->
<s:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    close="handleCloseEvent();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            // Handle the close button and Cancel button.
            private function handleCloseEvent():void {
                PopUpManager.removePopUp(this);
            }

            // Handle the OK button.
            private function processLogin():void {
                // Check credentials (not shown) then remove pop up.
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
    <s:Form>
        <s:FormItem label="User Name">
            <s:TextInput id="username" width="100%"/>
        </s:FormItem>
        <s:FormItem label="Password">
            <s:TextInput id="password"
                displayAsPassword="true"
                width="100%"/>
        </s:FormItem>
    </s:Form>
    <s:HGroup>
        <s:Button label="OK"
            click="processLogin();" />
        <s:Button label="Cancel"
            click="handleCloseEvent();"/>
    </s:HGroup>
</s:TitleWindow>
```

This file, named MyLoginForm.mxml, defines a TitleWindow container by using the `<s:TitleWindow>` tag. The TitleWindow container defines a Form container with two TextInput controls for user name and password. It also defines two Button controls for submitting the form and for closing the TitleWindow container. This example does not include the code for verifying the user name and password in the `submitForm()` event listener.

In this example, you process the form data in an event listener of the MyLoginForm.mxml component. To make this component more reusable, you can define the event listeners in your main application. This lets you create a generic form that leaves the data handling to the application that uses the form.

Close icon

By default, the TitleWindow container displays a close icon (a small *x* in the upper-right corner of the TitleWindow title bar) to make it appear similar to dialog boxes in a GUI environment.

Close event

The TitleWindow broadcasts a `close` event when the user clicks the close icon. Flex does not close the window automatically. Create a handler for that event and close the TitleWindow from within that event handler. In the MyLoginForm.mxml component, you use the same event handler for the close and Cancel buttons. You can create a skin file for the TitleWindow to hide the close button, or set the visible property of the close button to `false`.

Using the PopUpManager.createPopUp() method to create the pop-up Spark TitleWindow

To create a pop-up window, you call the PopUpManager `createPopUp()` method and pass it the parent, the name of the class that creates the pop-up, and the modal Boolean value. The following main application code creates the TitleWindow container defined in the previous section:

```
<?xml version="1.0"?>
<!-- containers\spark\SparkMainMyLoginForm.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import spark.components.TitleWindow;
            import myComponents.MyLoginForm;
            // Open the pop-up window.
            private function showLogin():void {
                // Create a non-modal TitleWindow container.
                var helpWindow:TitleWindow=
                    PopUpManager.createPopUp(this, MyLoginForm, false) as TitleWindow;
                PopUpManager.centerPopUp(helpWindow);
            }
        ]]>
    </fx:Script>

    <s:VGroup width="300" height="300">
        <s:Button label="Login"
            click="showLogin();"/>
    </s:VGroup>
</s:Application>
```

In this example, when the user selects the Login button, the event listener for the `click` event uses the `createPopUp()` method to create a TitleWindow container, passing to it the name of the MyLoginForm.mxml file as the class name.

By casting the return value of the `createPopUp()` method to TitleWindow, you can manipulate the properties of the pop-up TitleWindow container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\spark\SparkMainMyLoginFormCast.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
          import mx.managers.PopUpManager;
          import spark.components.TitleWindow;
          import myComponents.MyLoginForm;
          private function showLogin():void {
            // Create the TitleWindow container.
            var helpWindow:TitleWindow =
          PopUpManager.createPopUp(this, MyLoginForm, false) as TitleWindow;
            // Add title to the title bar.
            helpWindow.title="Enter Login Information";
            // Make title bar slightly transparent.
            helpWindow.setStyle("borderAlpha", 0.9);

            // Hide the close button.
            helpWindow.closeButton.visible = false;
          }
        ]]>
    </fx:Script>

    <s:VGroup width="300" height="300">
        <s:Button click="showLogin();" label="Login"/>
    </s:VGroup>
</s:Application>
```

Using the PopUpManager.addPopUp() method to create the pop-up Spark TitleWindow

You can use the `addPopUp()` method of the PopUpManager to create a pop-up window without defining a custom component. This method takes an instance of any class that implements IFlexDisplayObject. Because it takes a class instance, not a class, you can use ActionScript code in an `<fx:Script>` block to create the component instance to pop up, rather than as a separate custom component.

Using the `addPopUp()` method may be preferable to using the `createPopUp()` method if you have to pop up a simple dialog box that is not reused elsewhere in the application. However, it is not the best coding practice if the pop-up is complex, or if you want to create a component that can be used anywhere in the application.

The following example creates a pop-up with `addPopUp()` method and adds a Button control to that window that closes the window when you click it:

```
<?xml version="1.0"?>
<!-- containers\spark\MyPopUpButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600" width="600" >
    <fx:Script>
        <![CDATA[
            import spark.components.TitleWindow;
            import flash.events.*;
            import mx.managers.PopUpManager;
            import spark.components.Button;
            import mx.core.IFlexDisplayObject;

            // The variable for the TitleWindow container
            public var myTitleWindow:TitleWindow = new TitleWindow();
            // Method to instantiate and display a TitleWindow container.
            // This is the initial Button control's click event handler.
            public function openWindow(event:MouseEvent):void {
                // Set the TitleWindow container properties.
                myTitleWindow = new TitleWindow();
                myTitleWindow.title = "My Window Title";
                myTitleWindow.width= 220;
                myTitleWindow.height= 150;
                // Call the method to add the Button control to the
                // TitleWindow container.
                populateWindow();
                // Use the PopUpManager to display the TitleWindow container.
                PopUpManager.addPopUp(myTitleWindow, this, true);
            }

            // The method to create and add the Button child control to the
            // TitleWindow container.
            public function populateWindow():void {
                var btn1:Button = new Button();
                btn1.label="close";
                btn1.addEventListener(MouseEvent.CLICK, closeTitleWindow);
                myTitleWindow.addElement(btn1);
            }

            // The method to close the TitleWindow container.
            public function closeTitleWindow(event:MouseEvent):void {
                PopUpManager.removePopUp(myTitleWindow);
            }
        ]]>
    </fx:Script>
    <s:Button label="Open Window" click="openWindow(event);"/>
</s:Application>
```

Using the mouseDown event to close the Spark TitleWindow

You can use the mouseDownOutside event to close the pop-up window when a user clicks the mouse outside the TitleWindow. To do this, you add an event listener to the TitleWindow instance for the mouseDownOutside event, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\spark\MainMyLoginFormMouseDown.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import spark.components.TitleWindow;
            import myComponents.MyLoginForm;
            import mx.events.FlexMouseEvent;
            private var helpWindow:TitleWindow;
            private function showLogin():void {
                // Create the TitleWindow container.
                helpWindow = PopUpManager.createPopUp(this,
                    MyLoginForm, false) as TitleWindow;
                helpWindow.addEventListener("mouseDownOutside", removeForm);
            }
            private function removeForm(event:FlexMouseEvent):void {
                PopUpManager.removePopUp(helpWindow);
            }
        ]]>
    </fx:Script>

    <s:VGroup width="300" height="300">
        <s:Button click="showLogin();" label="Login"/>
    </s:VGroup>
</s:Application>
```

You define the event listener in the main application, and then assign it to the pop-up window when you create it. This technique lets you use a generic pop-up window, defined by the component MyLoginForm.mxml, and then modify the behavior of the component by assigning event listeners to it from the main application.

Creating a modal pop-up window

The `createPopUp()` method takes an optional *modal* parameter. You can set this parameter to `true` to make the window modal. When a TitleWindow is modal, you cannot select any other component while the window is open. The default value of *modal* is `false`.

The following example creates a modal pop-up window:

```
var pop1:IFlexDisplayObject = PopUpManager.createPopUp(this, MyLoginForm, true);
```

Dragging the popup Spark TitleWindow

After you pop up the Spark TitleWindow container, you can drag it around the application window. To drag the TitleWindow, click and hold the mouse in the title bar area of the window, then move the mouse.

By default, you click in the title bar area of the TitleWindow container to drag it. That area is defined by the `moveArea` skin part. You can create a custom skin class to modify the move area.

Events during a move operation

The Tile Window dispatches the following events as part of a move operation:

Event	**Descriptions**
`windowMove`	Dispatched when you stop dragging the window. This event can be dispatched multiple times before the `windowMoveEnd` event if you stop dragging the window but do not release the mouse button.
`windowMoveEnd`	Dispatched when the user releases the mouse button to end the drag.
`windowMoveStart`	Dispatched when the user presses the mouse button while the pointer is in the move area to begin the drag. This event is cancelable.
`windowMoving`	Dispatched repeatedly as the user drags the window. This event is cancelable.

The event handlers for these events receive an object of type TitleWindowBoundsEvent. The TitleWindowBoundsEvent class defines the `beforeBounds` and `afterBounds` properties, of type Rectangle. You can use these properties to determine the position of the TitleWindow as it is dragged, as the following table shows:

Event	**beforeBounds**	**afterBounds**
`windowMove`	The previous bounds of the window.	The current bounds of the window.
`windowMoveEnd`	The starting bounds of the window.	The ending bounds of the window.
`windowMoveStart`	The starting bounds of the window.	null
`windowMoving`	The current bounds of the window.	The future bounds of the window if the window moves to the current cursor position.

The following example shows a TitleWindow component that displays each event in a TextArea control as it occurs:

```
<?xml version="1.0"?>
<!-- containers\spark\myComponents\MyLoginFormMoveEvents.mxml -->
<s:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="300" height="300"
    close="handleCloseEvent();"
    windowMove="windowMoveHandler(event);"
    windowMoveEnd="windowMoveEndHandler(event);"
    windowMoveStart="windowMoveStartHandler(event);"
    windowMoving="windowMovingHandler(event);">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import spark.events.TitleWindowBoundsEvent;
            import mx.managers.PopUpManager;
            // Handle the close button and Cancel button.
            private function handleCloseEvent():void {
                PopUpManager.removePopUp(this);
            }
            protected function windowMoveHandler(event:TitleWindowBoundsEvent):void
            {
                myTA.text += "windowMove event occurred" + "\n"
            }
            protected function windowMoveEndHandler(event:TitleWindowBoundsEvent):void {
                myTA.text += "windowMoveEnd event occurred" + "\n"
            }
            protected function windowMoveStartHandler(event:TitleWindowBoundsEvent):void {
                myTA.text += "windowMoveStart event occurred" + "\n"
            }
            protected function windowMovingHandler(event:TitleWindowBoundsEvent):void {
                myTA.text += "windowMoving event occurred" + "\n"
            }
        ]]>
    </fx:Script>
    <s:TextArea id="myTA"
        width="100%" height="100%"/>

    <s:HGroup>
        <s:Button label="Cancel"
            click="handleCloseEvent();"/>
        <s:Button label="Clear"
                  click="myTA.text='';"/>
    </s:HGroup>
</s:TitleWindow>
```

The following example uses this component. As you drag the TitleWindow container around the application, the event handlers write a string to the TextArea control:

```
<?xml version="1.0"?>
<!-- containers\spark\SparkMainMoveEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import spark.components.TitleWindow;
            import myComponents.MyLoginFormMoveEvents;
            // Open the pop-up window.
            private function showLogin():void {
                // Create a non-modal TitleWindow container.
                var helpWindow:TitleWindow =
            PopUpManager.createPopUp(this, MyLoginFormMoveEvents, false) as TitleWindow;
                PopUpManager.centerPopUp(helpWindow);
            }
        ]]>
    </fx:Script>

    <s:VGroup width="300" height="300">
        <s:Button label="Login"
            click="showLogin();"/>
    </s:VGroup>
</s:Application>
```

You can disable dragging of the TitleWindow by setting the `visible` property of the `moveArea` skin part to `false`:

```
helpWindow.moveArea.visible = false;
```

Passing data to and from a Spark pop-up window

To make your pop-up window more flexible, you might want to pass data to it or return data from it. To do this, use the following guidelines:

- Create a custom component to be your pop-up. In most circumstances, this component is a TitleWindow container.
- Declare variables in your pop-up that you set in the application that creates the pop-up.
- Cast the pop-up to be the same type as your custom component.
- Pass a reference to that component to the pop-up window, if the pop-up window is to set a value on the application or one of the application's components.

For example, the following application populates a ComboBox in the pop-up window with an ArrayList defined in the main application.

```
<?xml version="1.0"?>
<!-- containers\spark\SparkMainArrayEntryForm.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            import mx.managers.PopUpManager;
            import myComponents.ArrayEntryForm;

            public function displayForm():void {
                /* ArrayList with data for the custom control ComboBox control. */
                var doctypes:ArrayList = new ArrayList(["*.as", "*.mxml", "*.swc"]);

                /* Create the pop-up and cast the
                   return value of the createPopUp()
                   method to the ArrayEntryForm custom component. */
                var pop1:ArrayEntryForm = ArrayEntryForm(
                    PopUpManager.createPopUp(this, ArrayEntryForm, true));

                /* Set TitleWindow properties. */
                pop1.title="Select File Type";

                /* Set properties of the ArrayEntryForm custom component. */
                pop1.targetComponent = ti1;
                pop1.myArray = doctypes;
                PopUpManager.centerPopUp(pop1);
            }
        ]]>
    </fx:Script>
    <s:TextInput id="ti1" text=""/>
    <s:Button id="b1" label="Select File Type"
        click="displayForm();"/>
</s:Application>
```

When creating the pop-up, the application casts the pop-up to be of type ArrayEntryForm, which is the name of the custom component that defines the pop-up window. If you do not do this, the application cannot access the properties that you create.

The application passes a reference to the TextInput component in the Application container to the pop-up window so that the pop-up can write its results back to the container. The application also passes the ArrayList of filename extensions for the pop-up ComboBox control's data provider, and sets the pop-up window's title. By setting these in the application, you can reuse the pop-up window in other parts of the application without modification, because it does not have to know the name of the component it is writing back to or the data that it is displaying, only that its data is in an Array and it is writing to a TextArea.

The following custom component, ArrayEntryForm.mxml, declares two variables. The first one is for the ArrayList that the parent application passes to the pop-up window. The second holds a reference to the parent application's TextInput control. The component uses that reference to update the parent application:

```
<?xml version="1.0"?>
<!-- containers\spark\myComponents\ArrayEntryForm.mxml -->
<s:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="200" borderAlpha="1"
    close="removeMe();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import spark.components.TextInput;
            import mx.managers.PopUpManager;
            import mx.collections.ArrayList;

            // Variables whose values are set by the main application.
            // Data provider array for the component's ComboBox control.
            [Bindable]
            public var myArray:ArrayList;
            // A reference to the TextInput control
            // in which to put the result.
            public var targetComponent:TextInput;
            // OK button click event listener.
            // Sets the target component in the application to the
            // selected ComboBox item value.
            private function submitData():void {
                targetComponent.text = String(cb1.selectedItem);
                removeMe();
            }
            // Cancel button click event listener.
            private function removeMe():void {
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
    <s:ComboBox id="cb1" dataProvider="{myArray}"/>
    <s:HGroup>
        <s:Button label="OK" click="submitData();"/>
        <s:Button label="Cancel" click="removeMe();"/>
    </s:HGroup>
</s:TitleWindow>
```

From within a pop-up custom component, you can also access properties of the parent application by using the `parentApplication` property. For example, if the application has a Button control named b1, you can get the label of that Button control, as the following example shows:

```
myLabel = parentApplication.b1.label;
```

This technique, however, uses a hard-coded value in the pop-up component for both the target component id in the parent and the property in the component.

Passing data using events

The following example modifies the example from the previous section to use event listeners defined in the main application to handle the passing of data back from the pop-up window to the main application. This example shows the ArrayEntryFormEvents.mxml file with no event listeners defined within it.

```
<?xml version="1.0"?>
<!-- containers\spark\myComponents\ArrayEntryFormEvents.mxml -->
<s:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="200"
    borderAlpha="1">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import mx.collections.ArrayList;

            // Variables whose values are set by the main application.
            // Data provider array for the component's ComboBox control.
            [Bindable]
            public var myArray:ArrayList;
        ]]>
    </fx:Script>
    <s:ComboBox id="cb1" dataProvider="{myArray}"/>
    <s:HGroup>
        <s:Button id="okButton" label="OK"/>
        <s:Button id="cancelButton" label="Cancel"/>
    </s:HGroup>
</s:TitleWindow>
```

The main application defines the event listeners and registers them with the controls defined within the pop-up window:

```
<?xml version="1.0"?>
<!-- containers\spark\SparkMainArrayEntryFormEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            import mx.managers.PopUpManager;
            import myComponents.ArrayEntryFormEvents;

            public var pop1:ArrayEntryFormEvents;

            public function displayForm():void {
                /* ArrayList with data for the custom control ComboBox control. */
                var doctypes:ArrayList = new ArrayList(["*.as", "*.mxml", "*.swc"]);

                /* Create the pop-up and cast the return value
                   of the createPopUp() method to the ArrayEntryFormEvents custom
                   component. */
                pop1 = ArrayEntryFormEvents(
                PopUpManager.createPopUp(this, ArrayEntryFormEvents, true));
                /* Set TitleWindow properties. */
                pop1.title="Select File Type";

                /* Set the event listeners for
                   the ArrayEntryFormEvents component. */
                pop1.addEventListener("close", removeMe);
                pop1.cancelButton.addEventListener("click", removeMe);
                pop1.okButton.addEventListener("click", submitData);

                /* Set properties of the ArrayEntryFormEvents custom control. */
                pop1.myArray = doctypes;
                PopUpManager.centerPopUp(pop1);
            }

            /* OK button click event listener.
               Sets the target component in the application to the
               selected ComboBox item value. */
            private function submitData(event:Event):void {
                ti1.text = String(pop1.cb1.selectedItem);
                removeMe(event);
            }
            /* Cancel button click event listener. */
            private function removeMe(event:Event):void {
                PopUpManager.removePopUp(pop1);
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:TextInput id="ti1" text=""/>
    </s:VGroup>
    <s:Button id="b1" label="Select File Type" click="displayForm();"/>
</s:Application>
```

The Spark Form, Spark FormHeading, and Spark FormItem containers

The Spark Form is a highly customizable container that supports multiple form designs and helps you create richer form experiences. When designing forms, you sometimes have limited space to display all the form elements. Spark forms help you solve this problem by providing horizontal and stacked layouts, as well as multiple customized columns.

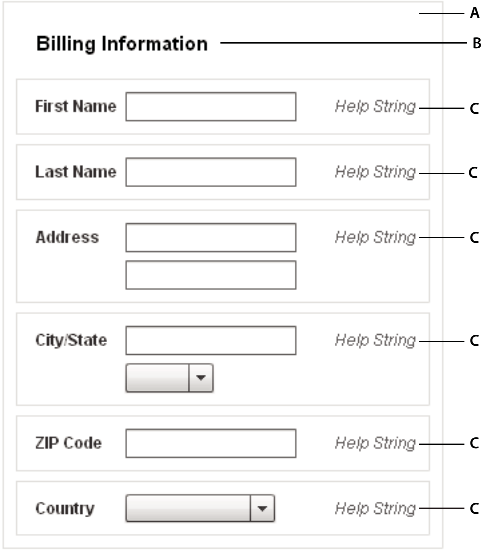
Spark Forms let you customize the appearance of your form through skinning.

Adobe Product Evangelist James Ward provides an overview of the Spark Form container in the video Overview of Spark Forms in Flex.

Creating a Spark Form

You use the `<s:Form>` tag to define a Spark Form. Specify an `id` value if you intend to refer to the entire form elsewhere in your MXML, either in another tag or in an ActionScript block.

To create a Spark Form, you typically use three different components: the Form container, FormHeading controls, and FormItem containers, as the following example shows:



Spark Form example
***A.** Form container* ***B.** FormHeading control* ***C.** FormItem containers*

For complete reference information, see Form, FormHeading, and FormItem in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

For a video on creating a Spark Form, see www.adobe.com/go/learn_flexsparkformvideo_en

Creating a Spark FormHeading control

A Spark FormHeading control specifies an optional label for a group of FormItem containers. The label is aligned with the start of the FormItem label controls in the form. You can have multiple FormHeading controls in your form to designate multiple content areas. You can also use FormHeading controls with a blank `label` property to create vertical space in your form.

You use the `<s:FormHeading>` tag to define a FormHeading container. Specify an `id` value if you intend to refer to the heading elsewhere in your MXML, either in another tag or in an ActionScript block.

The following code example defines the FormHeading control for the Spark Form:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\layouts\spark\SparkFormHeadingSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Form id="myForm" width="400" height="100">
        <s:FormHeading label="Spark Form Heading" />
        <!--Define FormItem containers here. -->
    </s:Form>
</s:Application>
```

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a Spark FormItem container

A Spark FormItem consists of the following parts:

- A single label

- A sequence label
- One or more child controls or containers, such as text input or button control
- Help content that provides a description of the form item or instructions for filling it out
- Required indicator to indicate if a form item has to be filled

By default, all the FormItem elements are arranged in a horizontal layout with the label placed on the left and the Help content on the right.

The following code example defines the Spark FormItem container for the Spark Form:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\layouts\spark\SparkFormItemSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Form id="myForm" width="450" height="125">
        <s:FormHeading label="Spark Form Heading" />
        <s:FormItem label="Username:">
            <s:TextInput id="username" />
            <s:helpContent>
                <s:Label text="Enter your LDAP username" />
            </s:helpContent>
        </s:FormItem>
    </s:Form>
</s:Application>
```

For complete reference information, see Spark FormItem in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Using sequence labels in a Spark FormItem control

You sometimes need to fill the form in a sequential order. Sequential labels help you design forms that define a visual order to the form items. When laying out form items in a Spark Form, you can define sequence labels using the `sequenceLabel` property.

The following code example uses sequence labels, incrementing the sequence number by one for each label:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\layouts\spark\SparkFormItemSequenceLabels.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Form id="myForm" width="350" height="325">
        <!-- Reduce gap between form items. -->
        <s:layout>
            <s:FormLayout gap="-14"/>
        </s:layout>
        <s:Label fontSize="16" text="My Spark Form"/>
        <s:FormItem label="Address" sequenceLabel="1.">
            <s:TextInput width="100%"/>
            <s:TextInput width="100%"/>
            <s:TextInput width="100%"/>
        </s:FormItem>
        <s:FormItem label="City" sequenceLabel="2.">
            <s:TextInput width="100%"/>
        </s:FormItem>
        <s:FormItem label="State" sequenceLabel="3.">
            <s:TextInput width="100%"/>
        </s:FormItem>
        <s:FormItem label="ZipCode" sequenceLabel="4.">
            <s:TextInput width="100%"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

You can customize the appearance of sequence labels by styling the FormItem container's sequenceLabelDisplay skin part. Properties like label `color` and `fontWeight` are explicitly set in the default FormItemSkin. To change these properties, you can create a custom skin using MXML or ActionScript and override the default styles.

Defining Help content in a Spark FormItem control

Help content provides a description for the form item or instructions for filling out the form item. When you create a FormItem, you use the `helpContent` property to define the Help content for the FormItem.

In a BasicLayout Group container, the Help content is placed next to the FormItem input control, by default. If you have multiple items in your helpContent group, you can place the Help content in an HGroup container or a VGroup container. The HGroup container lays out Help content horizontally and the VGroup container lays out Help content vertically.

The following code example defines Help content for FormItems in a Spark Form:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\layouts\spark\SparkFormItemHelpContent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600">

    <s:Form id="form1" x="164" y="38">
        <!-- Reduce space between form items. -->
        <s:layout>
            <s:FormLayout gap="-14"/>
        </s:layout>
        <s:FormHeading label="Mailing Information"/>
        <s:FormItem label="Address">
            <s:helpContent>
                <s:Label text="(eg. 123 Main Street)"/>
            </s:helpContent>
            <s:TextInput/>
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="City/State">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="Phone" fontFamily="Verdana" required="true">
            <s:helpContent>
                <s:VGroup>
                    <s:Label text="(xxx)-xxx-xxxx"/>
                    <s:Label text="Will appear in your profile"/>
                </s:VGroup>
            </s:helpContent>
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

You can customize the appearance of the Help content by styling the FormItem container's `helpContentGroup` skin part. Properties like `fontStyle` and `color` are explicitly set in the default FormItem skin. To change these properties, you can create a custom skin using MXML or ActionScript and override the default styles.

Spark Form and Spark FormItem skin classes

The Flex framework provides two sets of skin classes for the Spark Form and Spark FormItem controls:

The default skin classes define a horizontal layout of form items. The default skin classes include the following:

- FormHeadingSkin: A horizontal layout for a FormHeading.
- FormItemSkin: A horizontal layout for a FormItem with four columns and five skin parts. For more information, see "Using the required indicator for form items" on page 462.
- FormSkin: A four-column skin for the Form, where the columns are arranged horizontally in a single row.

The stacked skin classes define each form item with the label above the form control to use less horizontal space. The stacked skin classes include the following:

- StackedFormHeadingSkin: A stacked layout for a FormHeading
- StackedFormItemSkin: A stacked layout for a FormItem with four columns and five skin parts.
- StackedFormSkin: A three-column skin with two rows per item, and the label placed in the top row.

The following code example defines a Spark Form control using a StackedFormHeadingSkin, StackedFormItemSkin, and StackedFormSkin class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkFormStackedSkinClasses.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="300">

    <fx:Style>
       @namespace s "library://ns.adobe.com/flex/spark";

       s|Form
           {
               skinClass: ClassReference("spark.skins.spark.StackedFormSkin");
           }
       s|FormHeading
           {
               skinClass: ClassReference("spark.skins.spark.StackedFormHeadingSkin");
           }
       s|FormItem
           {
               skinClass: ClassReference("spark.skins.spark.StackedFormItemSkin");
           }
    </fx:Style>

    <s:Form id="frm" width="300">
        <!-- Reduce gap between form items. -->
        <s:layout>
            <s:FormLayout gap="-14"/>
        </s:layout>
        <s:FormHeading label="STACKED SPARK FORM" backgroundColor="Gray" color="#FFFFFF"
width="300"/>
        <s:FormItem sequenceLabel="1." label="First name" required="true">
            <s:TextInput id="fName" maxChars="64" />
        </s:FormItem>
        <s:FormItem sequenceLabel="2." label="Last name">
            <s:TextInput id="lName" maxChars="64" />
        </s:FormItem>
        <s:FormItem sequenceLabel="3." label="Address">
            <s:TextInput id="AddressLine1" maxChars="32"/>
            <s:TextInput id="AddressLine2" maxChars="32"/>
            <s:TextInput id="AddressLine3" maxChars="32"/>
        </s:FormItem>
        <s:FormItem sequenceLabel="4." label="City">
            <s:TextInput id="City" maxChars="64" />
        </s:FormItem>
        <s:FormItem sequenceLabel="5." label="State">
            <s:TextInput id="State" maxChars="64" />
        </s:FormItem>
        <s:FormItem sequenceLabel="6." label="ZipCode" required="true">
            <s:TextInput id="ZipCode" maxChars="64" />
        </s:FormItem>
    </s:Form>
</s:Application>
```

Using the required indicator for form items

The `required` state indicates that you must enter the FormItem's content before submitting the form. The `required` state occurs when you set the FormItem's `required` property to true as follows:

```
<s:FormItem label="Username:" required="true">
    <s:TextInput id="username" />
</s:FormItem>
```

When you set the `required` property to `true`, an asterisk (*) is used to denote the field. A Spark FormItem lets you customize the appearance of the required fields through skinning. For example, a required field can have a light green background, a bold label, or a "(required)" text.

An error state highlights the input error in the form item. The `error` state occurs when the FormItem's content fails validation. For example, you can define a red border or a light red background to indicate an error state. For more information on including form validation in your Spark Form, see “Validating data in a Spark Form” on page 462.

Validating data in a Spark Form

You can include form validation in your Spark Form. You can add custom logic to validate the input controls to ensure that the required data is entered in a proper format. To do so, you create the proper validator or formatter controls, and associate them with their Form Item controls. When a user enters incorrect data in the Form Item, the Form Item skin uses a different state to represent the erroneous state of the Form Item.

The following code example shows a Spark Form where the phone entry TextInput control is bound to a PhoneFormatter. The TextInput control is bound to the PhoneFormatter such that the data entered is formatted to display, as required. The ZIP code text input is bound to a ZipCodeValidator for data validation, such that, on entering incorrect data, an appropriate error message appears.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkFormValidation.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Declarations>
        <mx:PhoneFormatter
            id="pnf"
            areaCode="-1"
            areaCodeFormat="(###)"
            formatString="(###) ###-####"
            validPatternChars="+()#-. "/>
        <mx:ZipCodeValidator
            id="zcv"
            source="{ti_zip}"
            property="text"/>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            public var statesDP:ArrayCollection = new ArrayCollection
                    (["Arizona","California","Kansas","New Mexico","Texas","Wyoming"]);

            import mx.events.ValidationResultEvent;
            private var vResult:ValidationResultEvent;
            // Event handler to validate and format input.
            private function format():void {
                ti_phone.text = pnf.format(ti_phone.text);
            }
        ]]>
    </fx:Script>

    <s:Form width="500" height="600" backgroundColor="0xeeeeee">
        <s:Label fontSize="16" text="My Spark Form"/>
        <s:FormItem label="Address" sequenceLabel="1.">
            <s:TextInput width="100%"/>
            <s:TextInput width="100%"/>
            <s:TextInput width="100%"/>
            <s:helpContent>
                <s:Label text="(ex. 123 Main Street)" baseline="24" />
                <s:Button label="?" width="30" baseline="24" x="120"/>
            </s:helpContent>
        </s:FormItem>
        <s:FormItem label="City" sequenceLabel="2.">
            <s:TextInput width="100%"/>
        </s:FormItem>
        <s:FormItem label="State" sequenceLabel="3.">
            <s:ComboBox dataProvider="{statesDP}" width="100%"/>
        </s:FormItem>
        <s:FormItem label="ZipCode" sequenceLabel="4." required="true">
            <s:TextInput widthInChars="4" restrict="0123456789"/>
            <s:helpContent>
              <s:Label text="Will appear in your profile" left="0" right="0" baseline="24" />
            </s:helpContent>
        </s:FormItem>
        <s:FormItem id="phoneZipItem" label="Phone and Zip" sequenceLabel="5.">
            <s:TextInput id="ti_phone" width="100%" focusOut="format()"/>
            <s:TextInput id="ti_zip" widthInChars="4" restrict="0123456789"/>
            <s:helpContent>
                <s:Label text="(xxx)-xxx-xxxx" left="0" right="0" baseline="24" />
            </s:helpContent>
        </s:FormItem>
        <s:Label text="PhoneZip ErrorString : {phoneZipItem.elementErrorStrings}"/>
    </s:Form>
</s:Application>
```

Scrolling a Spark Form

The simplest way to add scrolling to a Spark Form is to wrap the Form in a Group container. Then wrap the Group container in a Scroller, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\layouts\spark\SparkFormScroll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:Scroller width="300" height="200">
        <s:Group>
            <s:Form>
                <s:FormItem label="Label 1:" textAlign="right">
                    <s:layout>
                        <s:HorizontalLayout/>
                    </s:layout>
                    <s:TextInput width="30"/>
                    <s:TextInput width="30"/>
                    <s:TextInput width="30"/>
                    <s:TextInput width="30"/>
                </s:FormItem>

                <s:FormItem label="Longer Label 2:" textAlign="right">
                    <s:layout>
                        <s:HorizontalLayout/>
                    </s:layout>
                    <s:TextInput width="30"/>
                    <s:TextInput width="30"/>
                    <s:TextInput width="30"/>
                    <s:TextInput width="30"/>
                </s:FormItem>

                <s:FormItem label="Label 3:" textAlign="right">
                    <s:TextInput width="200"/>
                </s:FormItem>
                <s:FormItem label="Label 4:" textAlign="right">
                    <s:TextInput width="200"/>
                </s:FormItem>
            </s:Form>
        </s:Group>
    </s:Scroller>
</s:Application>
```

Use a Group container because the Form container does not implements the spark.core.IViewport interface that is required by all children of Scroller.

Changing the layout of FormItems in a Spark Form

FormLayout is the layout class for the FormSkin class and StackedFormSkin class. FormLayout provides a vertical layout for its child FormItem containers in the Form. Elements using FormItemLayout within a FormLayout are aligned along columns.

Note: *Only use the FormLayout class with the Form container. Do not use it to lay out the contents of any other container.*

You can change the layout of a Spark Form by setting properties such as `gap` and padding properties on the FormLayout tag. The following example reduces the vertical space between the FormItems and indents the form by 25 pixels:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\layouts\spark\SparkFormLayout.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:Form id="form1" x="164" y="38">
        <!-- Reduce space between form items and indent form 25 pixels. -->
        <s:layout>
            <s:FormLayout gap="-14" paddingLeft="25"/>
        </s:layout>
        <s:FormHeading label="Mailing Information (Custom spacing)"/>
        <s:FormItem label="Address">
            <s:TextInput prompt="Enter your address"/>
        </s:FormItem>
        <s:FormItem label="City/State">
            <s:TextInput prompt="Enter your city and state"/>
        </s:FormItem>
        <s:FormItem label="Phone" fontFamily="Verdana" required="true">
            <s:TextInput prompt="Enter your phone number"/>
        </s:FormItem>
    </s:Form>
    <s:Form id="form2" x="164" y="38">
        <!-- Reduce space between form items. -->
        <s:layout>
            <s:FormLayout gap="0" paddingLeft="0"/>
        </s:layout>
        <s:FormHeading label="Mailing Information (Default properties)"/>
        <s:FormItem label="Address">
            <s:TextInput prompt="Enter your address"/>
        </s:FormItem>
        <s:FormItem label="City/State">
            <s:TextInput prompt="Enter your city and state"/>
        </s:FormItem>
        <s:FormItem label="Phone" fontFamily="Verdana" required="true">
            <s:TextInput prompt="Enter your phone number"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Using the FormItemLayout layout class

FormItemLayout is a grid layout for the FormItemSkin and StackedFormItemSkin classes. The FormItemLayout has a set of columns and rows, where each layout element can position its edges relative to each column and row.

The FormItemLayout class is necessary for the Form container to align the columns. Layout elements align to the specified columns and rows using constraints. For more information, see "Using constraint rows and columns with MX containers and the Spark FormLayout class" on page 387.

The Spark DataGroup and Spark SkinnableDataContainer containers

The Spark DataGroup and Spark SkinnableDataContainer containers take as children any components that implement the IVisualElement interface and are subclasses of DisplayObject. However, these containers are primarily used to take data items as children. Data items can be simple data items such String and Number objects, and more complicated data items such as Object and XMLNode objects.

An item renderer defines the visual representation of the data item in the container. The item renderer converts the data item into a format that can be displayed by the container. You must pass an item renderer to a DataGroup or SkinnableDataContainer container.

The main differences between the DataGroup and SkinnableDataContainer containers are:

- SkinnableDataContainer can be skinned. The DataGroup container is designed for simplicity and minimal overhead, and cannot be skinned.
- DataGroup can be a child of the Scroller control to support scroll bars. Create a skin for the SkinnableDataContainer to add scroll bars.

The default layout class of the DataGroup container is BasicLayout. The default layout class of the SkinnableDataContainer class is VerticalLayout. For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a Spark DataGroup and Spark SkinnableDataContainer container

You use the `<s:DataGroup>` and `<s:SkinnableDataContainer>` tags to define a DataGroup and SkinnableDataContainer container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The DataGroup and SkinnableDataContainer container are examples of data provider components. Data provider components require data for display or user interaction. To provide data, assign a collection which implements the IList interface, such as an ArrayList, ArrayCollection, or XMLListCollection object, to the container’s `dataProvider` property. For more information on using data providers, see “Data providers and collections” on page 898.

The `dataProvider` property takes an array of children, as the following example shows:

```
<s:DataGroup itemRenderer=...>
    <s:dataProvider>
        <mx:ArrayList>
            <fx:String>Dave Jones</fx:String>
            <fx:String>Mary Davis</fx:String>
            <fx:String>Debbie Cooper</fx:String>
        </mx:ArrayList>
    </s:dataProvider>
</s:DataGroup>
```

If you are using Flex Components as children of the container, you can specify them as the following example shows:

```
<s:DataGroup itemRenderer=...>
    <s:dataProvider>
        <mx:ArrayList>
            <s:Button/>
            <s:Button/>
            <s:Button/>
        </mx:ArrayList>
    </s:dataProvider>
</s:DataGroup>
```

Because dataProvider is the default property of the DataGroup and SkinnableDataContainer container, you do not have to specify a <s:dataProvider> child tag. Therefore, you can write the example as shown below:

```
<s:DataGroup itemRenderer=...>
    <mx:ArrayList>
        <fx:String>Dave Jones</fx:String>
        <fx:String>Mary Davis</fx:String>
        <fx:String>Debbie Cooper</fx:String>
    </mx:ArrayList>
</s:DataGroup>
```

You can mix different types of data items in a container, or mix data items and Flex components. For example, you might mix String, Object, and XML data in the same container. However, you must define an item renderer function to apply the correct item renderer to the child. For more information, see "Using an item renderer function with a Spark container" on page 481.

You can skin the SkinnableDataContainer in the same way that you skin the container. For an example of a skin, see "Creating a Spark SkinnableContainer container" on page 428.

Using a default item renderer with a Spark container

The DataGroup and SkinnableDataContainer containers require an item renderer to draw each container child on the screen. By default, the DataGroup and SkinnableDataContainer containers do not define an item renderer. You can configure the containers to use the item renderers provided by Flex, or define your own custom item renderer.

Flex ships with two item renderers: spark.skins.spark.

- spark.skins.spark.DefaultItemRenderer Converts its data item to a single String for display in a Spark Label control. It is useful when displaying a scalar data item, such as a String or a Number, that can be easily converted to a String.
- spark.skins.spark.DefaultComplexItemRenderer Displays a Flex component in a Group container. Each component is wrapped in its own Group container. Therefore, it is useful when the children of the container are visual elements, such as Flex components.

The following example uses the DefaultItemRenderer with a DataGroup container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerDefaultRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:DataGroup itemRenderer="spark.skins.spark.DefaultItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:String>Dave Jones</fx:String>
            <fx:String>Mary Davis</fx:String>
            <fx:String>Debbie Cooper</fx:String>
        </mx:ArrayList>
    </s:DataGroup>
</s:Application>
```

Each data item of the container is a String. Because you use the DefaultItemRenderer with the container, each String appears in the container in a Label control.

If the data item is of type Object or is a data type that is not easily converted to a String, then you either have to convert it to a String, or define a custom item renderer to display it. For more information, see “Passing data to a Spark item renderer” on page 474.

The following example shows a DataGroup container where all its children are Flex components. The DataGroup class uses the DefaultComplexItemRenderer to display each child:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleVisual.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:DataGroup itemRenderer="spark.skins.spark.DefaultComplexItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <s:Button/>
            <s:Button/>
            <s:Button/>
            <s:Button/>
        </mx:ArrayList>
    </s:DataGroup>
</s:Application>
```

Because you use the DefaultComplexItemRenderer with the container, each Button control appears in the container nested in its own Group container. By wrapping each control in a Group container, the item renderer can support selection highlighting for the individual children. However, if you do not want each control to appear in its own Group container, set the item renderer to `null`, as shown below:

```
<s:DataGroup itemRenderer="{null}">
```

***Note:** If you are only displaying visual elements in a DataGroup or SkinnableDataContainer container, you should instead use the Group or SkinnableContainer containers.*

You might be able to create your application by using just the DefaultItemRenderer and DefaultComplexItemRenderer classes. However, you typically define a custom item renderer if your data items are not simple values, or if you want more control over the appearance of your container children. For more information on creating a custom item renderer, see “Define a custom Spark item renderer” on page 470.

Adding and removing children at runtime

To modify the children of the DataGroup and SkinnableDataContainer containers at runtime, modify the `dataProvider` property. The following example uses event handlers to add and remove container children by calling the `addItem()` and `removeItemAt()` methods on the `dataProvider` property:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerAddRemoveChild.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[

            private function addDGChild():void {
                var newChild:String = "new child";
                myDG.dataProvider.addItem(newChild);

                addDG.enabled = false;
                removeDG.enabled = true;
            }

            private function removeDGChild():void {
                myDG.dataProvider.removeItemAt(3);

                addDG.enabled = true;
                removeDG.enabled = false;
            }
        ]]>
    </fx:Script>
    <s:DataGroup id="myDG"
        itemRenderer="spark.skins.spark.DefaultItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:String>Dave Jones</fx:String>
            <fx:String>Mary Davis</fx:String>
            <fx:String>Debbie Cooper</fx:String>
        </mx:ArrayList>
    </s:DataGroup>

    <s:Button id="addDG" label="Add Child"
        click="addDGChild();"/>
    <s:Button id="removeDG" label="Remove Child"
        enabled="false"
        click="removeDGChild();"/>
</s:Application>
```

For more information on using data providers, see “Data providers and collections” on page 898.

Using virtualization with Spark DataGroup and SkinnableDataContainer

A DataGroup or SkinnableDataContainer container can represent any number of children. However, each child requires an instance of an item renderer. If the container has many children, you might notice performance degradation as you add more children to the container.

Instead of creating an item renderer for each child, you can configure the container to use a virtual layout. With virtual layout, the container reuses item renderers so that it only creates item renderers for the currently visible children of the container. As a child is moved off the screen, possible by scrolling the container, a new child being scrolled onto the screen can reuse its item renderer.

To configure a container to use virtual layout, set the `useVirtualLayout` property to `true` for the layout associated with the container. Only the DataGroup or SkinnableDataContainer with the VerticalLayout, HorizontalLayout, and TileLayout supports virtual layout.

***Note:** If you define an itemRendererFunction to determine the item renderer for each data item, Flex will not reuse item renderers. The itemRendererFunction must examine each data item and create the item renderers as necessary for the specific data item type. While Flex does not reuse item renderers, it only creates enough item renderers for the currently visible data items. For more information, see "Creating a recyclable item renderer for virtual layout" on page 484.*

There are a few differences between the way a layout class works when virtual layout is enabled and when it is disabled:

- A layout with virtual layout enabled does not support the layout's major axis percent size property. This axis corresponds to the `percentHeight` property for the VerticalLayout class, and the `percentWidth` property for the HorizontalLayout class.
- A container using a virtual layout that contains few children whose sizes vary widely can respond poorly to interactive scrolling using the scroll thumb. No performance degradation occurs when scrolling using the scroll arrows or by clicking in the scroll track. Responsiveness improves as the variation in size decreases or the number of children increases.

Use virtual layout when the cost of creating or measuring a DataGroup is prohibitive because of the number of data elements or the complexity of the item renderers.

For more information on creating item renderers, see "Define a custom Spark item renderer" on page 470.

Custom Spark item renderers

The Spark list-based controls, such as List and ComboBox, support custom item renderers. You can also use Spark item renderers with some MX controls, such as the MX DataGrid and MX Tree controls.

Define a custom Spark item renderer

Several Spark components represent lists of items. These list-based components, such as DataGroup, List, and DataGrid, let the application user scroll through the item list. Some components, such as List, also let you select one or more items from the list.

You define a custom item renderer to control the display of a data item in a list-based component. The appearance can include the font, background color, border, and any other visual aspects of the data item.

An item renderer also defines the appearance of a data item when the user interacts with it. For example, the item renderer can display the data item one way when the user hovers over the data item, and in a different way when the user selects the data item.

Many Spark components support both skins and item renderers. While the item renderer defines the appearance of the data item, the skin defines the complete visual appearance of the component. The skin can include borders, scroll bars, and any other aspects of the appearance of the component. For more information on skins, see "Spark Skinning" on page 1602.

Adobe engineer Balaji Sridhar explains creating item renderers in Using Flash Builder 4 to create Spark item renderers .

Item renderer architecture

Create an item renderer in MXML or ActionScript. The advantage to creating item renderers in MXML is that it requires the least amount of code because much of the item renderer functionality is built into the base class, ItemRenderer, that you use to define MXML item renderers.

ActionScript item renderers provide the best performance because they give you complete control over the implementation. In an ActionScript item renderer, you only implement the code necessary to support your application requirements. Create an ActionScript item renderer as a subclass of the spark.components.LabelItemRenderer class for mobile applications, and the mx.core.UIComponent class for desktop applications.

The Spark item renderer architecture is defined by interfaces. Regardless of how you create your item renderer, MXML or ActionScript, the item renderer typically implements two interfaces:

Item renderer interface	Implemented by	Use
IDataRenderer	Item renderer	Defines the `data` property used to pass information to an item renderer. At a minimum, an item renderer must implement IDataRenderer to display data.
IItemRenderer	Item renderer	Defines the APIs that a component must implement to create an item renderer that can communicate with a host component to support user interaction with the data item. User interactions include selection, dragging, and the caret indicator. For some components, such as the Spark List control, user interaction includes item selection.

The list-based components that uses the item renderer is called the host component, or item renderer owner. To function as a host component, the component must implement the IItemRendererOwner interface.

Host component interface	Implemented by	Use
IItemRendererOwner	Host component of an item renderer	Defines the methods used by the host component to pass data to the item renderer.

Custom item renderer example

To better understand how item renderers work, examine the implementation of a custom item renderer. Shown below is the code for a custom MXML item renderer for the SkinnableDataContainer container that changes the font of the data item when the user hovers over the data item:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleCustomItemRenderer.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:states>
        <s:State name="normal"/>
        <s:State name="hovered"/>
    </s:states>

    <s:Label id="labelDisplay"
        verticalCenter="0"
        left="3" right="3" top="6" bottom="4"
        fontSize.hovered='14' fontStyle.hovered="italic"/>
</s:ItemRenderer>
```

The base class of this item renderer is the ItemRenderer class. The ItemRenderer class implements the IDataRenderer and IItemRenderer interfaces. It is also a subclass of the Group class, so it is itself a container. In the body of the item renderer, define the layout, view states, and child controls of the item renderer used to represent the data item.

The default layout of the ItemRenderer class is BasicLayout. In this example, since there is no specification for the `layout` property, the item renderer uses BasicLayout.

The item renderer can define view states. All view states are optional. In this example, you handle the normal and hovered view states. The normal state defines the appearance of the data item when there is no user interaction. The hovered view state defines the appearance when the user hovers the pointer over the data item.

This example uses the hovered view state to change the font when the user mouses over the data item. In this example, the custom item renderer displays the text in 14 point, italic font on mouse over. For more information on using view states in an item renderer, see "Defining item renderer view states for a Spark container" on page 488.

The Label control is centered vertically in the display area of the item renderer, and is constrained to be three pixels in from the left border, three pixels in from the right border, six pixels in from the top border, and four pixels in from the bottom border. You can use these same settings in your custom item renderers to mimic the look of the default Flex item renderer, or change them as necessary for your application.

The `id` of the Label control in the item renderer is `labelDisplay`. This is a specially named component in an item renderer. Flex writes the String representation of the data item to the component named `labelDisplay`. Flex also uses the `labelDisplay` component to determine the value of the `baselinePosition` property in the host component.

The following application uses this custom item renderer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleIR.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableDataContainer
        itemRenderer="myComponents.MySimpleCustomItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:String>Bill Smith</fx:String>
            <fx:String>Dave Jones</fx:String>
            <fx:String>Mary Davis</fx:String>
            <fx:String>Debbie Cooper</fx:String>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

Differences between mobile and desktop item renderers

An item renderer is often designed to appear differently in a mobile application than in a desktop application. Therefore, while it is possible, you typically do not use the same item renderer in a mobile application and in a desktop application.

For example, the differences between an item renderer for a mobile and a desktop application include the following:

- The size of a mobile item renderer tends to be larger than for a desktop item renderer.
- An item renderer for the selected item on in a mobile application usually indicates selection with a checkmark or other icon. For a desktop application, item renderers usually show selection by changing the `backgroundColor` style.
- The caret in a mobile application is usually indicated by changing the `backgroundColor` style. For a desktop application, the caret is usually indicated by drawing a border around the item.

The way you create item renderers can also depend on whether the application is for a desktop application or for a mobile application. You define item renderers in two ways:

- MXML

  Use MXML to define item renderers for desktop applications. MXML item renderers are simple to implement, but do not provide the highest performance often necessary for mobile applications. You typically create an MXML item renderer as a subclass of the spark.components.ItemRenderer class.

  For a mobile application, if your component hosting the item renderer displays only has a few data items or does not support scrolling, you might be able to use an MXML item renderer.

- ActionScript

  Use ActionScript for item renderers for applications that require the highest performance. For item renderers used in mobile applications, or for item renderers to use in both mobile and desktop applications, create them in ActionScript. For a mobile project, you typically create ActionScript item renderers as a subclass of the spark.components.LabelItemRenderer class.

Blogger Nahuel Foronda created a series of articles on Mobile ItemRenderer in ActionScript.

Blogger Rich Tretola created a cookbook entry on Creating a List with an ItemRenderer for a mobile application.

Interacting with a mobile and desktop item renderer

The mx.core.UIComponent class defines the `interactionMode` style property that you use to configure components for the type of input used in the application. For the Halo and Spark themes, the default value is `mouse` to indicate that the mouse is the primary input device. For the Mobile theme, the default value is `touch` to indicate the primary input is the touch screen.

One important difference between item renderers for desktop and for mobile applications is the way you interact with the application. For desktop applications, you use a mouse to interact with the application. You can use the mouse to:

- Move the mouse pointer over a data item. This is referred to as hovering over the item.
- Select the item by clicking on it. Many Spark components let the user select multiple data items by using the Shift and Control keys in combination with the mouse.
- Display a caret item. The caret item is the data item that currently has focus. The caret item can be the currently selected data item, or it can be a different item.
- Drag and drop data items when enabled by the Spark control.

When you interact with an application on a mobile device, you often use a touch screen or a five-way navigation control, not a mouse. With a mobile application, the data items in a list-based control do not enter the hovered state, and you typically do not have to support drag and drop. Therefore, when implementing an item renderer for mobile applications, you do not have to worry about supporting those situations.

MXML item renderers define view states. View states provide MXML item renderers with a simple mechanism for handling the different states of the data item. In ActionScript item renderers, you do not use view states. Instead, you handle user interaction in the body of your item renderer.

Differences between down and selected item renderers in a mobile application

For mobile applications, an item renderer must distinguish between an item being in the down state and the item being in the selected state. An item is down when the user presses down on the item. An item is selected when the user releases their finger after pressing down on the item.

***Note:** In a desktop application, one where `interactionMode` is `mouse`, selection occurs on the `mouseDown` event.*

When a user presses down on an item, the item renderer does not immediately enter the down state. Instead, the item renderer waits for the duration of time specified by the `touchDelay` property. The default delay duration specified by the `touchDelay` property is 100 ms.

This delay ensures that the user did not intend to initiate a scroll operation. If the item renderer immediately went to the down state, and the user intended to scroll, the item renderer would flicker from the normal state to the down state, and then back to the normal state. For more information on initiating a scroll operation in a mobile application, see Use scroll bars in a mobile application.

Working with item renderers

Passing data to a Spark item renderer

The host component of the item renderer is called the item renderer's owner. The host component passes information to the item renderer by using the properties defined by the IDataRenderer and IItemRenderer interfaces.

The following table describes the properties of the IDataRenderer and IItemRenderer interfaces:

Item renderer property	Type	Description	Interface
`data`	Object	The data item to render or edit as defined in the data provider of the host component.	IDataRenderer
`dragging`	Boolean	Contains `true` if the item renderer is being dragged. Mobile applications do not support drag and drop. Therefore, for an item renderer used only in a mobile application, you can define this property with a setter/getter that always returns `false`. To use the item renderer in a desktop application that supports drag and drop, you must implement it.	IItemRenderer
`itemIndex`	int	The index of the item in the data provider of the host component of the item renderer.	IItemRenderer
`label`	String	A String representation of the data item to display in the item renderer.	IItemRenderer
`selected`	Boolean	Contains `true` if the item renderer can show itself as selected. Many Spark components, such as the DataGroup and SkinnableDataContainers support item renderers, but do not support the selection of a data item. Other Spark components,. such as List, ComboBox, and DataGrid, support the selection of a data item.	IItemRenderer
`showsCaret`	Boolean	Contains `true` if the item renderer can show itself as focused.	IItemRenderer

Your item renderer might also implement the `hovered` property. The hovered property is not defined by an interface but is implemented by many Flex item renderers. The following table describes the `hovered` property:

Item renderer property	Type	Description	Interface
`hovered`	Boolean	Contains `true` when the user hovers over the list item. Typically, the item renderer listens for the `rollOver` and `rollOut` events to set `hovered` and to update the display of the item renderer accordingly. Mobile applications do not support hovered. For an item renderer used in a mobile application, you are not required to implement the hovered property. To use the item renderer in a desktop application, you must implement it. The predefined mobile item renderers supplied with Flex implement the hovered property so that you can use the item renderer in a mobile and desktop application.	None

The UIComponent class, the base class for all Flex components including item renderers, defines the `owner` property. The `owner` property contains a reference to the component that hosts the item renderer. For example, the SkinnableDataContainer can be the owner of an item renderer. From within the item renderer, you can access the host component by using the `owner` property.

The host component of an item renderer must implement the IItemRendererOwner interface. That interface defines the following methods to write information to the item renderer:

Host component method	Description	Interface
`itemToLabel()`	Converts the data item to a String representation. Host components can override this method to customize the String conversion.	IItemRendererOwner
`updateRenderer()`	Write the data item as a String to the `label` property. Updates the `owner` property with a reference to the host component. The last thing this method does is set the `data` property of the item renderer. Host components can override this method to write additional information to the item renderer.	IItemRendererOwner

Before you create a custom item renderer, decide how to pass the data item to the item renderer. In some situations, you want the host component to perform any processing on the data item before it passes it to the item renderer. If so, override the `itemToLabel()` and `updateRenderer()` methods in the host component. The item renderer then accesses the data by using the `label` property.

Instead of the host component processing the data item, the item renderer can perform the processing. If you want the item renderer to process the data item, use `IDataRenderer.data` property to pass it. The item renderer then accesses the `data` property and performs any processing on the data item before displaying it.

For an example that overrides the `itemToLabel()` and `updateRenderer()` methods, see "Passing data using the IItemRenderer.label property" on page 476. For an example that overrides the `data` property, see "Passing data using the IIDataRenderer.data property" on page 479.

Passing data using the IItemRenderer.label property

If the data item is a String, or a value that can easily be converted to a String, you can use the `IItemRenderer.label` property to pass it to the item renderer. If the data item is in a format that must be converted to a String representation, override the `IItemRendererOwner.itemToLabel()` method in the host component to customize the conversion.

***Note:** The example item renderers in this section are written in MXML. For examples written in ActionScript, see "Create a Spark item renderer in ActionScript" on page 495.*

In the following example, the children of the SkinnableDataContainer container are Strings specifying different colors:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataContainerColor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.*">
    <s:SkinnableDataContainer
        itemRenderer="myComponents.MySimpleColorRenderer">
        <mx:ArrayList>
            <fx:String>red</fx:String>
            <fx:String>green</fx:String>
            <fx:String>blue</fx:String>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

This example uses a custom item renderer named MySimpleColorRenderer, defined in the file MySimpleColorRenderer.mxml, that shows the String text in a background of the matching color. Shown below is the MXML item renderer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleColorRenderer.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    autoDrawBackground="false">

    <fx:Script>
        <![CDATA[

            // Property to hold the RGB color value.
            [Bindable]
            public var myColor:uint;

            // Write String to labelDisplay component.
            override public function set label(value:String):void
            {
                super.label = value;
                labelDisplay.text = label;

                // Determine the RGB color value from the data item.
                if (label == "red")
                    myColor = 0xFF0000;
                if (label == "green")
                    myColor = 0x00FF00;
                if (label == "blue")
                    myColor = 0x0000FF;
            }
        ]]>
    </fx:Script>

    <!-- Set the background color to the RGB color value.-->
    <s:Rect width="100%" height="100%" alpha="0.5">
        <s:fill>
            <s:SolidColor color="{myColor}" />
        </s:fill>
    </s:Rect>
    <!-- Display the color name -->
    <s:Label id="labelDisplay"/>
</s:ItemRenderer>
```

In this example, the item renderer overrides the `label` property to write the color to the Label control, and set the fill color of the Rect component.

This item renderer displays the data with no visual changes based on state. Therefore, it sets the `ItemRenderer.autoDrawBackground` property to `false`. This item renderer is useful for displaying data in the container when it does not have any user interaction. For an example of an item renderer that changes its display based on user interaction, see “Controlling the background color using the autoDrawBackground property” on page 487.

If you want to modify the String passed to the `label` property, you can override the `itemToLabel()` method in the host component. The `itemToLabel()` method has the following signature:

```
itemToLabel(item:Object):String
```

The method takes a single argument representing the data item. It returns a String representation of the data item for display in the item renderer.

In the following example, each data item is represented by an Object containing three fields. The custom SkinnableDataContainer container, called MyDataGroup, overrides the `itemToLabel()` method to format the Object as a String before passing the String to the item renderer. The example then uses the DefaultItemRenderer to display the text:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerOverride.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComps="myComponents.*">

    <!-- Define a custom DataGroup container to override the itemToLabel() method. -->
    <MyComps:MyDataGroup itemRenderer="spark.skins.spark.DefaultItemRenderer">
        <MyComps:layout>
            <s:VerticalLayout/>
        </MyComps:layout>

        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayList>
    </MyComps:MyDataGroup>
</s:Application>
```

The MyDataGroup.mxml file defines the custom SkinnableDataContainer container that overrides the `itemToLabel()` method:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MyDataGroup.mxml -->
<s:SkinnableDataContainer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Override to return the Object as a formatted String.
            override public function itemToLabel(item:Object):String {
                var tempString:String;
                if (item == null)
                    return " ";
                tempString = item.firstName + " " + item.lastName
                    + " " + ", ID: " + item.companyID;
                return tempString;
            }
        ]]>
    </fx:Script>
</s:SkinnableDataContainer>
```

Passing data using the IIDataRenderer.data property

Rather than processing the data in the host component, you can let the item renderer perform all the processing of the data item for display. In this situation, use the `ItemRenderer.data` property to pass the data item to the item renderer. This technique lets you define a set of item renderers that display the same data in different ways depending on your application requirements.

***Note:** The example item renderers in this section are written in MXML. For examples written in ActionScript, see "Create a Spark item renderer in ActionScript" on page 495.*

In the next example, each data item is represented by an Object that defines three fields:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableDataContainer
        itemRenderer="myComponents.MySimpleItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

The SkinnableDataContainer uses a custom item renderer named MySimpleItemRenderer.mxml. The custom item renderer displays the firstName and lastName fields in a single Label control, and display the companyID in a second Label control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRenderer.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:HGroup verticalCenter="0" left="2" right="2" top="2" bottom="2">
        <s:Label text="{data.lastName}, {data.firstName}"/>
        <s:Label text="{data.companyID}"/>
    </s:HGroup>
</s:ItemRenderer>
```

The `data` property contains an Object passed from the DataGroup container. The Object represents the data item in its original form. The renderer uses data binding to populate the controls in the item renderer from the `data` property. The two Label controls are defined in a Group container so that they can be layed out horizontally.

Rather than using data binding, you can override the `data` property in the item renderer. Within the override, you can modify the data or perform other processing, then set properties in the renderer. The following example shows an alternative implementation of MySimpleItemRenderer.mxml that overrides the `data` property:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererDataOverride.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[

            override public function set data(value:Object):void {
                super.data = value;

                // Check to see if the data property is null.
                if (value== null)
                    return;
                // If the data property is not null,
                // set the Label controls appropriately.
                nameLabel.text = value.firstName + ', ' + value.lastName;
                compLabel.text = value.companyID;
            }
        ]]>
    </fx:Script>
    <s:HGroup verticalCenter="0" left="2" right="2" top="2" bottom="2">
        <s:Label id="nameLabel"/>
        <s:Label id="compLabel"/>
    </s:HGroup>
</s:ItemRenderer>
```

Controlling the background display of an item renderer

An item renderer controls the background display of the data item in the host component. Typically, an item renderer changes the background display of the data item to reflect user interaction. For example, an item renderer can use a transparent background when there is no user interaction with the data item. It could draw a blue background when the data item is selected, and a different background when the user hovers over the data item.

Flex defines the following CSS styles to define background colors for the three common user interactions:

Interaction	Style	Value for Spark theme	Value for Mobile theme
none	`contentBackgroundColor`	0xFFFFFF (white)	0x464646 (dark grey)
hovered	`rollOverColor`	0xCEDBEF (light blue)	0xCEDBEF (light blue/grey)
selected	`selectionColor`	0xA8C6EE (dark blue)	0xB2B2B2 (light grey)

These colors are defined in the default.css file for the spark.swc and mobilecomponents.swc file. If you want your item renderer to mimic the color setting of Flex, use these same colors for setting the background color of your custom item renderers.

You can use these same styles in a custom item renderer to mimic the default background colors used by Flex component. Or, you can define your item renderers to use different colors and backgrounds.

If you define an MXML item renderer, you typically create it as a subclass of the ItemRenderer class. The ItemRenderer class defines the default background color for all types of user interactions. By default the item renderer draws a transparent background around the item when there is not user interaction. It draws a light-blue background around an item when you hover over it. For more information, see “Controlling the background color using the autoDrawBackground property” on page 487.

If you define an item renderer in ActionScript, you define the background colors of the data item. You can choose to implement your item renderer to use the predefined CSS styles, or define your own.

In your main application, you can redefine the CSS styles for the background colors. The following applications sets the `rollOverColor` style of the application to green:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleIRStyled.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|ItemRenderer { rollOverColor : green }
    </fx:Style>

    <s:SkinnableDataContainer
        itemRenderer="myComponents.MySimpleCustomItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:String>Bill Smith</fx:String>
            <fx:String>Dave Jones</fx:String>
            <fx:String>Mary Davis</fx:String>
            <fx:String>Debbie Cooper</fx:String>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

Using an item renderer function with a Spark container

In some applications, you display different types of data items in a single container. In this scenario, each type of data item needs its own item renderer. Or, you mix data items and Flex components in a container. To mix data items and Flex components, define different item renderers for the data items and the Flex components.

You can use an item renderer function to examine each data item to determine which item renderer to use. The `DataGroup.itemRendererFunction` and `SkinnableDataContainer.itemRendererFunction` property takes a function with the following signature:

```
function itemRendererFunction(item:Object):ClassFactory
```

Where `item` is the data item, and the return value is the item renderer. If the child is a Flex component, return DefaultComplexItemRenderer to display the child in a Group container. Alternatively, return `null` to display the child with no renderer.

The following example defines an item renderer function to return one item renderer for data items defined as Object, and another item renderer for data items defined as String:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerFunction.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[

            import myComponents.MySimpleItemRendererFunction;
            import spark.skins.spark.DefaultItemRenderer;

            private function selectRenderer(item:Object):ClassFactory {
                var classFactory:ClassFactory;
                if (item is String) {
                    // If the item is a String, use DefaultItemRenderer.
                    classFactory = new ClassFactory(DefaultItemRenderer);
                }
                else {
                    // If the item is an Object, use MySimpleItemRendererFunction.
                    classFactory = new ClassFactory(MySimpleItemRendererFunction);
                }
                return classFactory;
            }
        ]]>
    </fx:Script>
    <s:DataGroup itemRendererFunction="selectRenderer">
        <s:layout>
            <s:TileLayout requestedColumnCount="3"/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:String>617-555-1212</fx:String>
            <fx:String>Newton</fx:String>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:String>617-555-5555</fx:String>
            <fx:String>Newton</fx:String>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:String>617-555-6666</fx:String>
            <fx:String>Newton</fx:String>
        </mx:ArrayList>
    </s:DataGroup>
</s:Application>
```

You also use item renderer function when mixing data items and Flex components in the container. Flex components implement the IVisualElement interface, and therefore do not need an item renderer to draw them on the screen. In your item renderer function, you can determine if the data item corresponds to a Flex component. If so, the item renderer function returns the DefaultComplexItemRenderer, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerFunctionVisual.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.core.IVisualElement;

            import myComponents.MySimpleItemRendererEmployee;
            import spark.skins.spark.DefaultComplexItemRenderer;

            private function selectRenderer(item:Object):ClassFactory {
                var classFactory:ClassFactory;
                if(item is IVisualElement){
                    // If the item is a Flex component, use DefaultComplexItemRenderer.
                    classFactory = new ClassFactory(DefaultComplexItemRenderer);
                }
                else if (item is Object){
                    // If the item is an Object, use MySimpleItemRendererFunction.
                    classFactory = new ClassFactory(MySimpleItemRendererEmployee);
                }
                return classFactory;
            }
        ]]>
    </fx:Script>
    <s:DataGroup itemRendererFunction="selectRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
            <s:Button label="Add Employee"/>
        </mx:ArrayList>
    </s:DataGroup>
</s:Application>
```

Spark item renderer precedence

The DataGroup and SkinnableDataContainer containers use the follow rules to determine the item renderer for a child:

1 If the `itemRendererFunction` property is defined, call the associated function to obtain the item renderer. If the function returns `null`, go to rule 2.

2 If the `itemRenderer` property is defined, use the specified item renderer to display the item.

3 If the item is implements mx.core.IVisualElement and is of type flash.display.DisplayObject, use it directly.

4 Dispatch a runtime error if no item renderer found.

Creating a recyclable item renderer for virtual layout

With virtual layout disabled, the DataGroup and SkinnableDataContainer containers create one instance of the item renderer for each child. With virtual layout enabled, the container only creates enough item renderers to display its currently visible children. Virtual layout greatly reduces the overhead required to use the DataGroup and SkinnableDataContainer containers.

With virtual layout enabled, when a child is moved off the visible area of the container, its item renderer is recycled. When the item renderer is reused, its `data` property is set to the data item representing the new child. Therefore, if a recycled item renderer performs any actions based on the value of the `data` property, it must first check that the property is not `null`.

When the item renderer is reassigned, Flex also calls the `updateRenderer()` method of the item renderer owner. This method must set the `owner` and `label` properties on the item renderer. Subclasses of SkinnableDataContainer, can use the `updateRenderer()` method to set additional properties on the item renderer.

Because a container can reuse an item renderer, ensure that you fully define its state. For example, you use a CheckBox control in an item renderer to display a `true` (checked) or `false` (unchecked) value based on the current value of the `data` property. A common mistake is to assume that the CheckBox control is always in its default state of unchecked and only inspect the data property for a value of `true`.

However, remember that the CheckBox can be recycled and had previously been checked. Therefore, inspect the `data` property for a value of `false`, and explicitly uncheck the control if it is checked, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererCB.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    dataChange="setMgr();">

    <fx:Script>
        <![CDATA[
            private function setMgr():void {
                // Check to see if the data property is null.
                if (data == null)
                    return;
                // If the data property is not null,
                // set the CheckBox control appropriately..
                if (data.manager == "yes") {
                    mgr.selected = true;
                }
                else {
                    mgr.selected = false;
                }
            }
        ]]>
    </fx:Script>
    <s:HGroup verticalCenter="0" left="2" right="2" top="2" bottom="2">
        <s:Label text="{data.lastName}, {data.firstName}"/>
        <s:Label text="{data.companyID}"/>
        <s:CheckBox id="mgr"/>
    </s:HGroup>
</s:ItemRenderer>
```

Use the `dataChange` event of the ItemRenderer class to detect the change to its `data` property. This event is dispatched whenever the `data` property changes. Alternatively, you can override the `data` property.

Alternatively, you can override the `ItemRenderer.data` property itself, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererCBData.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[

            override public function set data(value:Object):void {
                super.data = value;
                // Check to see if the data property is null.
                if (value== null)
                    return;
                // If the data property is not null,
                // set the CheckBox control appropriately..
                if (value.manager == "yes") {
                    mgr.selected = true;
                }
                else {
                    mgr.selected = false;
                }
            }
        ]]>
    </fx:Script>
    <s:HGroup verticalCenter="0" left="2" right="2" top="2" bottom="2">
        <s:Label text="{data.lastName}, {data.firstName}"/>
        <s:Label text="{data.companyID}"/>
        <s:CheckBox id="mgr"/>
    </s:HGroup>
</s:ItemRenderer>
```

Defining a typical item for determining the size of an item renderer

When using virtual layout with the DataGroup and SkinnableDataContainer containers, you can pass to the container a data item that defines a typical data item. The container then uses the typical data item, and the associated item renderer, to determine the default size of the child. By defining the typical item, the container does not have to size each child as it is drawn on the screen.

If you do not specify the data item, by default the control uses the first item in the data provider as the typical data item.

Use the `typicalItem` property of the container to specify the data item, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerTypicalItem.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var typicalObj:Object = {
                    firstName:"Long first name",
                    lastName:"Even longer last name",
                    companyID:"123456",
                    manager:"yes"
            };
        ]]>
    </fx:Script>

    <s:Scroller>
        <s:DataGroup itemRenderer="myComponents.MySimpleItemRendererCB"
            height="100"
            typicalItem="{typicalObj}" >
            <s:layout>
                <s:VerticalLayout useVirtualLayout="true"/>
            </s:layout>
            <mx:ArrayList>
                <fx:Object firstName="Bill" lastName="Smith"
                    companyID="11233" manager="yes"/>
                <fx:Object firstName="Dave" lastName="Jones"
                    companyID="13455" manager="no"/>
                <fx:Object firstName="Mary" lastName="Davis"
                    companyID="11543" manager="yes"/>
                <fx:Object firstName="Debbie" lastName="Cooper"
                    companyID="14266" manager="no"/>
                <fx:Object firstName="Bob" lastName="Martins"
                    companyID="11233" manager="yes"/>
                <fx:Object firstName="Jack" lastName="Jones"
                    companyID="13455" manager="no"/>
                <fx:Object firstName="Sam" lastName="Johnson"
                    companyID="11543" manager="yes"/>
                <fx:Object firstName="Tom" lastName="Fitz"
                    companyID="14266" manager="no"/>
                <fx:Object firstName="Dave" lastName="Mead"
                    companyID="11233" manager="yes"/>
                <fx:Object firstName="Dave" lastName="Jones"
                    companyID="13455" manager="no"/>
                <fx:Object firstName="Mary" lastName="Davis"
                    companyID="11543" manager="yes"/>
                <fx:Object firstName="Debbie" lastName="Cooper"
                    companyID="14266" manager="no"/>
            </mx:ArrayList>
        </s:DataGroup>
    </s:Scroller>
</s:Application>
```

In this example, you define typicalObj, an Object that represents a data item with a long value for the firstName and lastName fields. You then pass typicalObj to the `typicalItem` property of the container. The container uses that data item, and the associated item renderer, to determine the size of the children.

Specifying a value for the `typicalItem` property passes that value, and the associated item renderer, to the `typicalLayoutElement` property of the layout of the container. For more information on the `typicalLayoutElement` property, see "Set the row height or column width of a layout" on page 423.

Create a Spark item renderer in MXML

MXML item renderers require the least amount of code to create. Typically, you create an MXML item renderer as a subclass of the ItemRenderer class.

Controlling the background color using the autoDrawBackground property

To completely control the background color of an item renderer, set the `ItemRenderer.autoDrawBackground` property to `false`. When you set that property to `false`, your item renderer is responsible for displaying the background colors for all user interactions.

The following custom item renderer alternates the background color between white and green for the items in a SkinnableDataContainer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MyAlternatingItemRenderer.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    autoDrawBackground="false">

    <fx:Script>
        <![CDATA[

            // Make the default background color white.
            [Bindable]
            public var myBGColor:int = 0xFFFFFF;

            // Override the itemIndex set function to draw a
            // white background behind even number items,
            // and a green background behind odd numbered items.
            override public function set itemIndex(value:int):void {
                if ((value%2) == 0) {
                    myBGColor= 0xFFFFFF;
                }
                if ((value%2) == 1) {
                    myBGColor= 0xCCFF66;
                }
            }
        ]]>
    </fx:Script>
    <s:states>
        <s:State name="normal"/>
        <s:State name="hovered"/>
    </s:states>
    <s:Rect id="myRect"
        left="0" right="0" top="0" bottom="0"
        alpha="1.0">
        <s:stroke>
            <s:SolidColorStroke
                color="0xA8C6EE"
                weight="1"/>
        </s:stroke>
        <s:fill>
            <!-- Bind the myBGColor property to the fill color. -->
            <s:SolidColor
                color="{myBGColor}"/>
        </s:fill>
    </s:Rect>

    <s:Label id="labelDisplay"
        verticalCenter="0"
        left="3" right="3" top="6" bottom="4"
        fontSize.hovered='14' fontStyle.hovered="italic"/>
</s:ItemRenderer>
```

This example overrides the `itemIndex` property of the ItemRenderer class. The `itemIndex` property contains the index of the data item in the data provider of the host component. In the override, set the background color to green for odd-numbered items, and to white for even-numbered items.

The following application uses this item renderer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerAlternatingIR.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableDataContainer
        itemRenderer="myComponents.MyAlternatingItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:String>Bill Smith</fx:String>
            <fx:String>Dave Jones</fx:String>
            <fx:String>Mary Davis</fx:String>
            <fx:String>Debbie Cooper</fx:String>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

Defining item renderer view states for a Spark container

Item renderers support optional view states. You typically use view states in MXML item renderers to control the appearance of a data item based on user interaction with the item. The ItemRenderer class supports all views states so that you can use those item renderers with list-based classes.

Flex defines the following view states that you can support in your item renderers.

- normal The data item has no user interaction.
- hovered The mouse is over the data item.
- selected The data item is selected.

- dragging The data item is being dragged.
- normalAndShowCaret The data item is in the normal state, and it has focus in the item list.
- hoveredAndShowCaret The data item is in the hovered state, and it has focus in the item list.
- selectedAndShowCaret The data item is in the normal state, and it has focus in the item list.

When the user interacts with a control in a way that changes the view state of the item renderer, Flex first determines if the renderer defines that view state. If the item renderer supports the view state, Flex sets the item renderer to use that view state. If the item renderer does not supports the view state, Flex does nothing.

The selected, normalAndShowCaret, hoveredAndShowCaret, and selectedAndShowCaret view states are supported by the list-based components. The list-based components are subclasses of the spark.components.supportClasses.ListBase class. The DataGroup and SkinnableDataContainer containers do not implement these view states.

The view states supported by your item renderers, and the action performed by a change to each view state, are determined by your application requirements. For example, you can alter the display of the data renderer based on the view state, show different data, or do nothing. Your item renderer can also define additional view states.

In the next example, you define a SkinnableDataContainer to display an Object with four fields: firstName, lastName, companyID, and phone:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleStates.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableDataContainer itemRenderer="myComponents.MySimpleItemRendererWithStates">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith"
                companyID="11233" phone="617-555-1212"/>
            <fx:Object firstName="Dave" lastName="Jones"
                companyID="13455" phone="617-555-1213"/>
            <fx:Object firstName="Mary" lastName="Davis"
                companyID="11543" phone="617-555-1214"/>
            <fx:Object firstName="Debbie" lastName="Cooper"
                companyID="14266" phone="617-555-1215"/>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

The following item renderer displays the text of the Label controls in bold, blue font for the hover state:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererWithStates.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    autoDrawBackground="false">
    <s:states>
        <s:State name="normal"/>
        <s:State name="hovered"/>
    </s:states>

    <s:HGroup verticalCenter="0" horizontalCenter="0">
        <s:Label text="{data.lastName}, {data.firstName}"
            color.hovered="blue"
            fontWeight.hovered="bold"/>
        <s:Label text="{data.companyID}"
            color.hovered="blue"
            fontWeight.hovered="bold"/>
        <s:Label text="{data.phone}"
            color.hovered="blue"
            fontWeight.hovered="bold"/>
    </s:HGroup>
</s:ItemRenderer>
```

Because a SkinnableDataContainer does not support the selected view state, the item renderer does not define any settings for the selected state.

You can also include a transition in an item renderer. A transition plays whenever you change a view state. The following item renderer uses a transition to display the company ID and telephone number of the employee on mouse over:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererWithTrans.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    autoDrawBackground="false">
    <s:states>
        <s:State name="normal"/>
        <s:State name="hovered"/>
    </s:states>
    <s:transitions>
        <s:Transition fromState="normal">
            <s:Sequence>
                <s:Resize target="{this}" />
                <mx:SetPropertyAction targets="{[cID, empPhone]}"
                    name="visible" value="true" />
             </s:Sequence>
        </s:Transition>
        <s:Transition toState="normal">
            <s:Sequence>
                <mx:SetPropertyAction targets="{[cID, empPhone]}"
                    name="visible" value="false" />
                <s:Resize target="{this}" />
            </s:Sequence>
        </s:Transition>
    </s:transitions>
    <s:VGroup verticalCenter="0" horizontalCenter="0">
        <s:Label text="{data.lastName}, {data.firstName}"
            color.hovered="blue"
            fontWeight.hovered="bold"/>
        <s:Label id="cID"
            includeIn="hovered"
            includeInLayout.hovered="true"
            includeInLayout.normal="false"
            text="{data.companyID}"/>
        <s:Label id="empPhone"
            includeIn="hovered"
            includeInLayout.hovered="true"
            includeInLayout.normal="false"
            text="{data.phone}"/>
    </s:VGroup>
</s:ItemRenderer>
```

The following application uses this item renderer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleStatesTransition.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableDataContainer itemRenderer="myComponents.MySimpleItemRendererWithTrans">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith"
                companyID="11233" phone="617-555-1212"/>
            <fx:Object firstName="Dave" lastName="Jones"
                companyID="13455" phone="617-555-1213"/>
            <fx:Object firstName="Mary" lastName="Davis"
                companyID="11543" phone="617-555-1214"/>
            <fx:Object firstName="Debbie" lastName="Cooper"
                companyID="14266" phone="617-555-1215"/>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

For more information on view states, see “View states” on page 1847. For more information on transitions, see “Transitions” on page 1870.

Defining an inline item renderer for a Spark container

The examples of item renderers shown above are all defined in an MXML file. That makes the item renderer highly reusable because you can reference it from multiple containers.

You can also define inline item renderers in the MXML definition of a component. By using an inline item renderer, your code can all be defined in a single file. However, it is not easy to reuse an inline item renderer.

The following example uses an inline item renderer with the SkinnableDataContainer container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerInline.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableDataContainer>
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayList>
        <s:itemRenderer>
            <fx:Component>
                <s:ItemRenderer>
                    <s:Group verticalCenter="0" left="2" right="2" top="2" bottom="2">
                        <s:layout>
                            <s:HorizontalLayout/>
                        </s:layout>
                        <s:Label text="{data.lastName}, {data.firstName}"/>
                        <s:Label text="{data.companyID}"/>
                    </s:Group>
                </s:ItemRenderer>
            </fx:Component>
        </s:itemRenderer>
    </s:SkinnableDataContainer>
</s:Application>
```

Notice that you define the item renderer inline by using the `itemRenderer` property of the DataGroup container. The first child tag of the `itemRenderer` property is always the `<fx:Component>` tag. Inside the `<fx:Component>` tag is the same code as was in the MySimpleItemRenderer.mxml file shown above.

Items allowed in an inline component

There is only one restriction on what you can and cannot do in an inline item renderer. You cannot create an empty `<fx:Component></fx:Component>` tag. For example, you can combine effect and style definitions in an inline item renderer along with your rendering logic.

You can include the following items in an inline item renderer:

- Binding tags
- Effect tags
- Metadata tags
- Model tags
- Scripts tags
- Service tags
- State tags
- Style tags
- XML tags

- id attributes, except for the top-most component

Using the Component tag

The <fx:Component> tag defines a new scope in an MXML file, where the local scope of the item renderer is defined by the MXML code block delimited by the <fx:Component> and </fx:Component> tags. To access elements outside the local scope of the item renderer, you prefix the element name with the outerDocument keyword.

For example, you define one variable named localVar in the scope of the main application. You define another variable with the same name in the scope of the item renderer. From within the item renderer, you access the application's localVar by prefixing it with outerDocument keyword, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerInlineScope.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            // Variable in the Application scope.
            [Bindable]
            public var localVar:String="Application scope";
        ]]>
    </fx:Script>
    <s:SkinnableDataContainer>
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayList>
        <s:itemRenderer>
            <fx:Component>
                <s:ItemRenderer>
                    <fx:Script>
```

```
                        <![CDATA[
                            // Variable in the Renderer scope.
                            [Bindable]
                            public var localVar:String="Renderer scope";
                        ]]>
                    </fx:Script>
                    <s:Group verticalCenter="0" left="2" right="2" top="2" bottom="2">
                        <s:layout>
                            <s:HorizontalLayout/>
                        </s:layout>
                        <s:Label text="{data.lastName}, {data.firstName}"/>
                        <s:Label text="{data.companyID}"/>
                        <s:Label text="{'Renderer localVar = ' + localVar}"/>
                       <s:Label text="{'Application localVar = ' + outerDocument.localVar}"/>
                    </s:Group>
                </s:ItemRenderer>
            </fx:Component>
        </s:itemRenderer>
    </s:SkinnableDataContainer>
</s:Application>
```

Creating a reusable inline item renderer

Rather than defining an inline item renderer in the definition of a component, you can define a reusable inline item renderer for use in multiple locations in your application.

To create a reusable inline item renderer, specify the `className` property of the `<fx:Component>` tag. By naming the class, you define a way to reference the item renderer, and the elements of the item renderer.

The following example uses the `<fx:Component>` tag to define an inline item renderer for two containers:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerInlineReuse.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:Component id="inlineRenderer" className="myIR">
            <s:ItemRenderer>
                <s:Group verticalCenter="0" horizontalCenter="0">
                    <s:layout>
                        <s:HorizontalLayout/>
                    </s:layout>
                    <s:Label text="{data.lastName}, {data.firstName}"/>
                    <s:Label text="{data.companyID}"/>
                </s:Group>
            </s:ItemRenderer>
        </fx:Component>
    </fx:Declarations>
    <s:SkinnableDataContainer itemRenderer="myIR">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
    <s:SkinnableDataContainer itemRenderer="myIR">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayList>
            <fx:Object firstName="Jim" lastName="Sullivan" companyID="11233"/>
            <fx:Object firstName="Joan" lastName="Connors" companyID="13455"/>
            <fx:Object firstName="Jack" lastName="Wilson" companyID="11543"/>
            <fx:Object firstName="Jeff" lastName="Lodge" companyID="14266"/>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:Application>
```

In this example, you use data binding to specify the renderer as the value of the `itemRenderer` property for the two containers.

Create a Spark item renderer in ActionScript

To ensure that your application achieves the highest performance, implement item renderers in ActionScript. Typically, you create item renderers in ActionScript only for mobile applications, but you can create them for desktop application as well.

The item renderer typically implements two interfaces:

Item renderer interface	Implemented by	Use
IDataRenderer	Item renderer	Defines the `data` property used to pass information to an item renderer. At a minimum, an item renderer must implement IDataRenderer to display data.
IItemRenderer	Item renderer	Defines the APIs that a component must implement to create an item renderer that can communicate with a host component to support user interaction with the data item. User interactions include selection, dragging, and the caret indicator. For some components, such as the Spark List control, user interaction includes item selection.

Create an ActionScript item renderer in one of the following ways:

- Create an ActionScript subclass from an existing item renderer class, such as spark.components.LabelItemRenderer, spark.components.IconItemRenderer, or spark.components.supportClasses.ItemRenderer. The existing item renderer classes already implement all necessary interfaces.

  For the highest performance in a mobile application, create a subclass of spark.components.LabelItemRenderer. In your subclass, you can control the background display and the layout of the data items.

  If you want to use the built-in Flex layout mechanism, and are not as concerned with performance, create a subclass of spark.components.supportClasses.ItemRenderer.

  For more information, see "Create an ActionScript item renderer as a subclass of the LabelItemRenderer class" on page 497.

  For more information on using LabelItemRenderer and IconItemRenderer, see "Using a mobile item renderer with a Spark list-based control" on page 524.

- Create an ActionScript subclass of the mx.core.UIComponent class.

  By implementing an item renderer as a subclass of the UIComponent class, you can obtain the best performance because you only have to implement the logic necessary to support your application. However, basing an item renderer on UIComponent requires the most effort. To be used as an item renderer, your subclass must implement the mx.core.IDataRenderer and spark.components.IItemRenderer interfaces.

For example implementations of item renderers written in ActionScript, view the source code for the spark.components.LabelItemRenderer, spark.components.IconItemRenderer, and spark.components.supportClasses.ItemRenderer classes.

Creating item renderers in ActionScript

To define an item renderer in ActionScript, override the necessary methods from the parent class, and call any necessary invalidation methods from within your overrides. For general information on this process, see "Create advanced Spark visual components in ActionScript" on page 2451.

The methods that you have to override in your class depend on the parent class of the item renderer. You optionally can override one or more of the following protected methods of the parent class:

Method	Description
`commitProperties()`	Commits any changes to component properties, either to make the changes occur at the same time or to ensure that properties are set in a specific order. For more information, see "Implementing the commitProperties() method for MX components" on page 2483.
`createChildren()`	Creates any child components of the component. For example, the ComboBox control contains a TextInput control and a Button control as child components. For more information, see "Implementing the createChildren() method for MX components" on page 2482.

Method	**Description**
`measure()`	Sets the default size and default minimum size of the component. For more information, see "Implementing the measure() method for MX components" on page 2485.
`styleChanged()`	Detects changes to style properties. For more information, see "Overriding the styleChanged() method" on page 2502.
`updateDisplayList()`	Sizes and positions the children of the component on the screen based on all previous property and style settings, and draws any skins or graphic elements used by the component. The parent container for the component determines the size of the component itself. ***Note:** You typically only have to implement this method when you use the mx.core.UIComponent class as the base class of your item renderer. You do not have to implement it when creating a subclass of the LabelItemRenderer class. Override* `LabelItemRenderer.drawBackground()` *and* `LabelItemRenderer.layoutContent()` *instead.* For more information, see "Implementing the updateDisplayList() method for MX components" on page 2489.

If the parent class is the LabelItemRenderer, you typically override one or both of the following methods:

Method	**Description**
`drawBackground()`	Defines the background display of the item renderer.
`layoutContents()`	Lays out the children of the item renderer.

Flex uses an invalidation mechanism to synchronize modifications to components. Flex implements the invalidation mechanism as a set of methods that you call to signal that something about the component has changed and requires Flex to call the component's `commitProperties()`, `measure()`, or `updateDisplayList()` methods.

The following table describes the invalidation methods:

Invalidation method	**Description**
`invalidateProperties()`	Marks a component so that its `commitProperties()` method gets called during the next screen update.
`invalidateSize()`	Marks a component so that its `measure()` method gets called during the next screen update.
`invalidateDisplayList()`	Marks a component so that its `updateDisplayList()` methods get called during the next screen update.

When a component calls an invalidation method, it signals to Flex that the component must be updated. When multiple components call invalidation methods, Flex coordinates the updates so that they all occur together during the next screen update. For more information on these methods, see "About the invalidation methods for MX components" on page 2476.

Create an ActionScript item renderer as a subclass of the LabelItemRenderer class

The following example shows a view component for a mobile application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\mobile\views\ListColor.mxml -->
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    title="Colors">
    <s:layout>
        <s:VerticalLayout paddingTop="10"/>
    </s:layout>

    <s:Label text="Favorites"/>
    <s:SkinnableDataContainer
        itemRenderer="myComponents.MySimpleColorRenderer">
        <mx:ArrayList>
            <fx:String>red</fx:String>
            <fx:String>green</fx:String>
            <fx:String>blue</fx:String>
        </mx:ArrayList>
    </s:SkinnableDataContainer>
</s:View>
```

In this example, the SkinnableDataContainer contains three data items corresponding to three colors. In this example, you define an item renderer in the file MySimpleColorRenderer.as. Your custom item renderer displays the text of the data item and changes the background color of the data item to match the text.

Because this is a mobile application, for optimal performance create the item renderer as a subclass of the LabelItemRenderer class. This example does not change the way the data item is layed out in the container, only its background color. Therefore, you only override the `LabelItemRenderer.drawBackground()` method, and not the `LabelItemRenderer.layoutContent()` method.

The MySimpleColorRenderer.as item renderer is shown below:

```
package myComponents
{
    // containers\mobile\myComponents\MySimpleColorRenderer .as
    import spark.components.LabelItemRenderer;

    public class MySimpleColorRenderer extends LabelItemRenderer
    {
        public function MySimpleColorRenderer() {
            super();
        }

        // Use the value of the myColor property to draw
        // the background color of the item in the list.
        override protected function drawBackground(unscaledWidth:Number,
unscaledHeight:Number):void {
            // Define a var to hold the color.
            var myColor:uint;
            // Determine the RGB color value from the label property.
            if (data == "red")
                myColor = 0xFF0000;
            if (data == "green")
                myColor = 0x00FF00;
            if (data == "blue")
                myColor = 0x0000FF;
            graphics.beginFill(myColor, 1);
            graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);

        }
    }
}
```

This item renderer accesses the `LabelItemRenderer.data` property in the override of the `drawbackground()` method to determine the background color of the item. The `label` property contains the String value to display in the item renderer.

Override the LabelItemRenderer.layoutComponents() method in an item renderer

By default, the LabelItemRenderer class uses a single spark.components.supportClasses.StyleableTextField control to display a String. The name of the StyleableTextField control in the item renderer is `labelDisplay`.

In the next example, you define an item renderer to display multiple fields of a data item in separate StyleableTextField controls in the item renderer. This example uses the predefined `labelDisplay` control to display the first and last name fields of the data item. It then adds a second StyleableTextField control to display the company ID field.

Shown below is the view component for the mobile application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\mobile\views\EmployeeMainViewIR.mxml -->
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    title="Employees View Main">
    <s:layout>
        <s:VerticalLayout paddingTop="10"/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function myLabelFunction(item:Object):String {
                return item.lastName + ', ' + item.firstName;;
            }
        ]]>
    </fx:Script>

    <s:Label text="Select an employee name"/>
    <s:List id="myList"
        width="100%" height="100%"
        itemRenderer="myComponents.MyGroupItemRenderer"
        labelFunction="myLabelFunction">
        <s:ArrayCollection>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </s:ArrayCollection>
    </s:List>
</s:View>
```

The List control uses the `itemRenderer` property to specify the name of the custom item renderer as MyGroupItemRenderer.as. In MyGroupItemRenderer.as, you:

- Define the StyleableTextField control named compLabelDisplay.
- Override the protected `createChildren()` method to create the compLabelDisplay control.
- Override the protected `styleChanged()` method to propagate and style changes to the compLabelDisplay control.
- Override the `data` property to initialize the compLabelDisplay control. The `data` property contains the original Object representing the data item in the List control.
- Override the `measure()` method to calculate the size of the labelDsiplay and compLabelDisplay controls.
- Override the `layoutContents()` method to size and position the labelDisplay and compLabelDisplay controls. The override uses the padding styles to calculate the availale space for the components.

Shown below is the definition of MyGroupItemRenderer, the item renderer:

```
package myComponents
{
    // containers\mobile\myComponents\MyGroupItemRenderer.as
    import spark.components.LabelItemRenderer;
    import spark.components.supportClasses.StyleableTextField;
    public class MyGroupItemRenderer extends LabelItemRenderer
    {
        public function MyGroupItemRenderer(){
            super();
        }

        // Define the StyleableTextField control used
        // to display the company ID.
        public var compLabelDisplay:StyleableTextField;

        // The distance between the label and the company ID.
        public var verticalGap:Number = 10;

        // Override createChildren() to create the StyleableTextField control.
        override protected function createChildren():void {
            super.createChildren();

            // Make sure it does not already exist.
            if (!compLabelDisplay) {
                compLabelDisplay = new StyleableTextField();

                // Specify The object that provides styles for the control.
                // This property must be set for the control to pick up
                // the correct styles.
                compLabelDisplay.styleName = this;

                // Set basic attributes of the control.
                compLabelDisplay.editable = false;
                compLabelDisplay.selectable = false;
                compLabelDisplay.multiline = false;
                compLabelDisplay.wordWrap = false;

                // Add the control as a child of the item renderer.
                addChild(compLabelDisplay);
            }
        }

        // Override styleChanged() to proopgate style changes to compLabelDisplay.
        override public function styleChanged(styleName:String):void {
            super.styleChanged(styleName);

            // Pass any style changes to compLabelDisplay.
            if (compLabelDisplay)
                compLabelDisplay.styleChanged(styleName);
        }

        // Override the data property to initialize compLabelDisplay.
        // The label function in the view specifies the String
        // displayed in labelDisplay.
        override public function set data(value:Object):void {
            super.data = value;
            compLabelDisplay.text = String(value.companyID);
        }

        // Override measure() to calculate the size required by the item renderer.
        override protected function measure():void {
            // Measure the labelDisplay by calling super.measure()
            super.measure();
            // Then consider the compLabelDisplay if it exists.
            if (compLabelDisplay)
            {
             var horizontalPadding:Number = getStyle("paddingLeft") + getStyle("paddingRight");
              var verticalPadding:Number = getStyle("paddingTop") + getStyle("paddingBottom");

                // Commit the styles changes to compLabelDisplay.
                // This method must be called before the text is displayed,
                // and any time the styles have changed.
                // This method does nothing if the styles have already been committed.
                compLabelDisplay.commitStyles();
                measuredWidth =  Math.max(getElementPreferredWidth(labelDisplay),
getElementPreferredWidth(compLabelDisplay))
                measuredWidth += horizontalPadding;

                measuredHeight =  getElementPreferredHeight(labelDisplay);
                measuredHeight += getElementPreferredHeight(compLabelDisplay);
                measuredHeight += verticalPadding + verticalGap;
            }
        }

        // Override layoutContents() to lay out the item renderer.
        override protected function layoutContents(unscaledWidth:Number,
unscaledHeight:Number):void {
            // Because you are handling the layout of both the
            // predefined labelDisplay component and the new
            // compLabelDisplay component, you do not have to call
            // super.layoutContents().

            // Make sure labelDisplay and compLabelDisplay exist.
            if (!labelDisplay)
                return;
            if (!compLabelDisplay)
                return;

            // Get the padding from the associated styles.
            var paddingLeft:Number = getStyle("paddingLeft");
            var paddingRight:Number = getStyle("paddingRight");
            var paddingTop:Number = getStyle("paddingTop");
            var paddingBottom:Number = getStyle("paddingBottom");

            // Calculate the available space for the component.
            var viewWidth:Number  = unscaledWidth - paddingLeft - paddingRight;
            var viewHeight:Number = unscaledHeight - paddingTop - paddingBottom;

            // Calcualte the size of the labelDisplay component.
            var labelWidth:Number = Math.max(viewWidth, 0);
            var labelHeight:Number = 0;

            if (label != "") {
                labelDisplay.commitStyles();

                // Reset text if it was truncated before.
                if (labelDisplay.isTruncated)
                    labelDisplay.text = label;

                labelHeight = getElementPreferredHeight(labelDisplay);
            }

            // Set the size and position of the labelDisplay component.
            setElementSize(labelDisplay, labelWidth, labelHeight);
            setElementPosition(labelDisplay, paddingLeft, paddingTop);

            // Attempt to truncate the text now that we have its official width
            labelDisplay.truncateToFit();

            // Size and position the compLabelDisplay component.
            var compLabelWidth:Number = Math.max(viewWidth, 0);
            var compLabelHeight:Number = 0;

            compLabelDisplay.commitStyles();
            compLabelHeight = getElementPreferredHeight(compLabelDisplay);

            setElementSize(compLabelDisplay, compLabelWidth, compLabelHeight);
            setElementPosition(compLabelDisplay, paddingLeft, paddingTop + labelHeight + 
verticalGap);
        }
    }
}
```

Using a Spark item renderer with an MX control

Many MX list-based controls, such as the DataGrid, AdvancedDataGrid, and Tree, support item renderers. These three MX list-based controls also support item editors. Item editors let you create a custom view of data during the editing process. The item editor returns the new data back to the control so that the control can update its data provider.

Flash Builder has built-in support for creating Spark item renderers. To support the use of Flash Builder for creating item editors and item editors for MX controls, Flex defines the mx.controls.listClasses.MXItemRenderer class as a subclass of the Spark spark.components.supportClasses.ItemRenderer class. You can use the MXItemRenderer class to create item renderers and item editors for the MX DataGrid and Tree classes.

***Note:** Many MX controls, such as the MX List and MX TileList controls, support item renderers or item editors. However, Flex provides the Spark List and Spark TileLayout class as replacements for the MX List and MX TileList controls. Always use the Spark components, when possible, in your application. There is no Spark equivalent for the MX DataGrid, MX AdvancedDataGrid, and MX Tree controls. Therefore, only those MX controls support the MXItemRenderer class.*

About MX item renderers and item editors

The MXItemRenderer class simplifies the development of item renderers and editors in Flash Builder. By basing item renderers and item editors on the MXItemRenderer class, your MX controls can take advantage of all of the features of the ItemRenderer class, including Flash Builder support.

All of the functionality of MX item renderers and item editors is supported by the MXItemRenderer class. Therefore, you should be familiar with the architecture of MX item renderers and editors before you create them based on the MXItemRenderer class.

See the following documentation for basic information on MX item renderers:

- “About MX item renderers” on page 1006
- “Creating MX item renderers and item editor components” on page 1033
- “About the MX cell editing process” on page 1048
- “About the MX cell editing process” on page 1048
- “Using item renderers with the AdvancedDataGrid control” on page 1442
- “Returning data from an MX item editor” on page 1049
- “Using MX cell editing events” on page 1057
- “MX item editor examples” on page 1062

Using the MXItemRenderer class

You base a Spark item renderer or item editor for an MX control on the MXItemRenderer class. While you can base an item renderer or editor on the MXItemRenderer class itself, you typically create them based on one of the following subclasses of MXItemRenderer:

- MXDataGridItemRenderer   The base class for a Spark item renderer or editor for the MX DataGrid control.
- MXAdvancedDataGridItemRenderer   The base class for a Spark item renderer or editor for the MX AdvancedDataGrid control.
- MXTreeItemRenderer   The base class for a Spark item renderer or editor for the MX Tree control.

Creating a Spark item renderer for an MX DataGrid control

The following application shows an MX DataGrid control that uses a custom item renderer:

```
<?xml version="1.0"?>
<!-- itemRenderers\sparkmx\SparkMainDGImageRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: '../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: '../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" height="400" width="600"
        dataProvider="{initDG}">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover"
                itemRenderer="myComponents.RendererDGImage"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you use the custom item renderer to display the album cover for each album listed in the DataGrid control. The Cover field of the data provider of the DataGrid contains the path to the JPG file for the cover.

Shown below is the definition of the RendererDGImage.mxml file that defines the item renderer:

```
<?xml version="1.0"?>
<!-- itemRenderers\sparkmx\myComponents\RendererDGImage.mxml -->
<s:MXDataGridItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">
        <mx:Image id="albumImage" height="175" source="{data.Cover}"/>
</s:MXDataGridItemRenderer>
```

Notice that this item renderer is based on the MXDataGridItemRenderer class. The item renderer uses the Image control to display the album cover.

The next example also displays the album name and cover image, but uses a single column of the DataGrid to display both fields:

```
<?xml version="1.0"?>
<!-- itemRenderers\sparkm\SparkMainDGTitleRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: '../../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: '../../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" width="600"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist" />
            <mx:DataGridColumn dataField="Album"
                itemRenderer="myComponents.RendererDGTitleImage" />
            <mx:DataGridColumn dataField="Price"  />
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

The RendererDGTitleImage.mxml file implements the render. It uses a Spark Label control and the MX Image control to display the album name and cover image in a single cell:

```
<?xml version="1.0"?>
<!-- itemRenderers\sparkmx\myComponents\RendererDGTitleImage.mxml -->
<s:MXDataGridItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:HorizontalLayout verticalAlign="middle"
            paddingLeft="5" paddingRight="5"/>
    </s:layout>
    <s:Label id="albumName"
        width="100%"
        text="{data.Album}"/>
    <mx:Image id="albumImage"
        source="{data.Cover}"/>
</s:MXDataGridItemRenderer>
```

Creating a Spark item renderer for an MX AdvancedDataGrid control

The following application uses the AdvancedDataGrid control:

```
<?xml version="1.0"?>
<!-- itemrenderers/sparkmx/SparkHierarchicalADGSimpleRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
              import mx.collections.ArrayCollection;

              include "SimpleHierarchicalData.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn id="diffCol"
                headerText="Difference"/>
        </mx:columns>
        <mx:rendererProviders>
            <mx:AdvancedDataGridRendererProvider column="{diffCol}"
                depth="3" renderer="myComponents.SummaryRenderer"/>
        </mx:rendererProviders>
    </mx:AdvancedDataGrid>
</s:Application>
```

This application uses the custom item renderer defined in the file SummaryRenderer.mxml to display the value of the Difference column. The value is determined by the difference between the Estimate and Actual columns. If the territory representative exceeded sales estimate, the cell appears in green. If the territory representative did not exceed the estimate, the cell appears in red.

The SummaryRenderer.mxml also includes a CurrencyFormatter to format the value in dollars. The SummaryRenderer.mxml is shown below:

```
<?xml version="1.0"?>
<!-- itemrenderers/sparkmx/myComponents/SummaryRenderer.mxml -->
<s:MXAdvancedDataGridItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    textAlign="center">

    <fx:Script>
        <![CDATA[
            override public function set data(value:Object):void
            {
                // Calculate the difference.
                var diff:Number =
                    Number(value["Actual"]) - Number(value["Estimate"]);
                if (diff < 0)
                {
                    // If Estimate was greater than Actual,
                    // display results in red.
                    setStyle("color", "red");
                    myLabel.text = "Undersold by " + usdFormatter.format(diff);
                }
                else
                {
                    // If Estimate was less than Actual,
                    // display results in green.
                    setStyle("color", "green");
                    myLabel.text = "Exceeded estimate by " + usdFormatter.format(diff);
                }
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:CurrencyFormatter id="usdFormatter" precision="2"
            currencySymbol="$" decimalSeparatorFrom="."
            decimalSeparatorTo="." useNegativeSign="true"
            useThousandsSeparator="true" alignSymbol="left"/>
    </fx:Declarations>
    <s:Label id="myLabel"/>
</s:MXAdvancedDataGridItemRenderer>
```

Creating a Spark item renderer for an MX Tree control

The following application uses an MX Tree control to display XML data:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- itemRenderers\tree\SparkMainTreeItemRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initCollections();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            import mx.collections.*;

            public var xmlBalanced:XMLList =
                <>
                    <node label="Containers">
                        <node label="DividedBoxClasses">
                            <node label="BoxDivider" data="BoxDivider.as"/>
                        </node>
                        <node label="GridClasses">
                            <node label="GridRow" data="GridRow.as"/>
                            <node label="GridItem" data="GridItem.as"/>
                            <node label="Other File" data="Other.as"/>
                        </node>
                    </node>
                    <node label="Data">
                        <node label="Messages">
                            <node label="DataMessage"
                                data="DataMessage.as"/>
                            <node label="SequenceMessage"
                                data="SequenceMessage.as"/>
                        </node>
                        <node label="Events">
                            <node label="ConflictEvents"
                                data="ConflictEvent.as"/>
                            <node label="CommitFaultEvent"
                                data="CommitFaultEvent.as"/>
                        </node>
                    </node>
                </>;

            [Bindable]
            public var xlcBalanced:XMLListCollection;

            private function initCollections():void {
                xlcBalanced = new XMLListCollection(xmlBalanced);
            }
        ]]>
    </fx:Script>
    <mx:Text width="400"
        text="The nodes with children are in bold red text, with the number of children in 
parenthesis.)"/>
    <mx:Tree id="compBalanced"
        width="400" height="500"
        dataProvider="{xlcBalanced}"
        labelField="@label"
        itemRenderer="myComponents.MyTreeItemRenderer"/>
</s:Application>
```

This application uses the item renderer defined by the MyTreeItemRenderer.mxml file to display the parent nodes in a red, bold font. It also shows the number of child nodes for each parent in parenthesis. Shown below is the definition of MyTreeItemRenderer.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- itemRenderers\sparkmx\myComponents\MyTreeItemRenderer.mxml -->
<s:MXTreeItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009" 
    xmlns:s="library://ns.adobe.com/flex/spark" 
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.controls.treeClasses.*;
            import mx.collections.*;

            // Override the set method for the data property
            // to set the font color and style of each node.
            override public function set data(value:Object):void {
                super.data = value;
                if(treeListData.hasChildren)
                {
                    setStyle("color", 0xff0000);
                    setStyle("fontWeight", 'bold');
                    var tmp:XMLList = 
                        new XMLList(treeListData.item);
                    var myStr:int = tmp[0].children().length();
                    labelDisplay.text =  treeListData.label + 
                        "(" + myStr + ")";
                }
                else
                {
                    setStyle("color", 0x000000);
                    setStyle("fontWeight", 'normal');
                    labelDisplay.text =  treeListData.label;
                }
            }
        ]]>
    </fx:Script>

    <s:HGroup left="0" right="0" verticalCenter="0">
        <s:Rect id="indentationSpacer"
            width="{treeListData.indent}" height="22"
            alpha="0">
            <s:fill>
                <s:SolidColor color="0xFFFFFF" />
            </s:fill>
        </s:Rect>
        <s:Group id="disclosureGroup">
            <s:BitmapImage source="{treeListData.disclosureIcon}"
                width="16" height="16"
                visible="{treeListData.hasChildren}" />
        </s:Group>
        <s:BitmapImage source="{treeListData.icon}"
            width="16" height="16"/>
        <s:Label id="labelDisplay" />
    </s:HGroup>
</s:MXTreeItemRenderer>
```

The item renderer uses the `treeListData` property of the MXTreeItemRenderer class to determine the information about the tree node. The `treeListData` property is of type TreeListData. TreeListData defines information such as the depth of the node in the tree, any icon associated with the node, and the data object of the node from the tree's data provider.

This item renderer overrides the `ItemRenderer.data` property to control the display of the tree nodes. The override first determines if the node is a parent node and, if so, sets the text display to use a bold, red font. It then adds the number of children to the display of the node.

The second part of the item renderer defines the appearance of the node in the Tree control. In this example, you use a Rect control to define a white background for the node. The item renderer uses two BitmapImage controls to define the appearance of any icons. The Label control defines the appearance of the text displayed by the node.

Creating a Spark item editor for an MX DataGrid control

An MX DataGrid, AdvancedDataGrid, or Tree control displays an item editor when the `editable` property of the control is set to `true`. The item editor appears when the user releases the mouse button while over a cell, tabs to the cell, or in another way attempts to edit the cell. By default, the `editable` property is `false`.

When creating an item editor for an MX DataGrid or Tree control, the item editor usually returns a single value corresponding to the new value in the control's data provider. For example, the following item renderer returns a single value named myRetValue:

```
<?xml version="1.0"?>
<!-- itemRenderers\sparkmx\myComponents\NSEditor.mxml -->
<s:MXDataGridItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            public var myRetVal:int = 0;
        ]]>
    </fx:Script>

    <!-- Use the valueCommit event when the user selects the
         cell but does not change the value. -->
    <s:NumericStepper id="myNS"
        value="{data.quant}"
        stepSize="1"
        maximum="50"
        change="myRetVal=myNS.value;"
        valueCommit="myRetVal=myNS.value;"/>
</s:MXDataGridItemRenderer>
```

This item editor contains a single NumericStepper control. When the item editor is open, the NumericStepper appears in the DataGrid control. The user can then use the NumericStepper to edit the value of the cell.

The user ends the editing session by removing focus from the cell. Flex then commits the new value in the DataGrid control. The following application uses this item editor:

```
<?xml version="1.0"?>
<!-- itemRenderers\sparkmx\SparkMainNSEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG" dataProvider="{myDP}"
            variableRowHeight="true"
            editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1"
                headerText="Order #"/>
            <mx:DataGridColumn dataField="quant"
                itemEditor="myComponents.NSEditor"
                editorDataField="myRetVal"/>
        </mx:columns >
    </mx:DataGrid>
</s:Application>
```

Notice in the main application that the DataGridColumn uses the `itemEditor` property to specify the name of the item editor. It uses the `editorDataField` property to specify the name of the property of the item editor that contains the return value.

Creating an inline Spark item editor for an MX DataGrid control

An MX DataGrid, AdvancedDataGrid, or Tree control can use an inline item renderer or editor. The following example uses a Spark DropDownList control as the item editor for a column of an MX DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\sparkmx\SparkDGInlineRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: '../assets/slanted.jpg', Rating:'none'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: '../assets/brighten.jpg', Rating:'none'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" height="400" width="600"
        dataProvider="{initDG}" editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Rating" editorDataField="rating">
                <mx:itemEditor>
                    <fx:Component>
                        <s:MXDataGridItemRenderer focusEnabled="true" height="22" >
                            <fx:Script>
                                <![CDATA[
                                    import mx.collections.ArrayList;
                                    public function get rating():String {
                                        return  dd.selectedItem;
                                    }
                                ]]>
                            </fx:Script>
                            <s:DropDownList id="dd" top="5" left="5"
                                selectedItem="{data.Rating}"
                                initialize="dd.dataProvider =
                                    new ArrayList(['none', 'no good', 'good', 'great'])"/>
                        </s:MXDataGridItemRenderer>
                    </fx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
            <mx:DataGridColumn dataField="Cover"
                itemRenderer="myComponents.RendererDGImage"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

This example adds a Rating column to the DataGrid. The user clicks in a cell of the Rating column to open the item editor. When the user removes focus from the cell, the value of the DropDownList is copied to the data provider of the DataGrid.

Spark list-based controls

The Spark list-based controls are designed to display data items in different visual formats and layouts.

Using Spark list-based controls

The Spark list-based controls include the following:

- ButtonBar
- ComboBox
- DataGrid
- DropDownList
- List
- TabBar

The Spark list-based controls are all subclasses of the SkinnableDataContainer and ListBase classes. The SkinnableDataContainer class is designed to display data items by using item renderers. Data items can be simple data types, such as String and Number objects, or more complex data types such as Object and XMLNode objects.

The ListBase class is a subclass of the SkinnableDataContainer class. The ListBase class adds support for item selection in the list of data items, and for controlling the text displayed in the list. The ButtonBar, ComboBox, DropDownList, List, and TabBar controls add additional functionality to the ListBase class.

For more information on SkinnableDataContainer, and on creating item renderers, see “The Spark DataGroup and Spark SkinnableDataContainer containers” on page 466.

More Help topics

"Sorting and matching" on page 2062

Defining the children of a list-based control

The list-based controls are examples of data provider components. Data provider components require data for display or user interaction. To provide data, assign a collection which implements the IList interface, such as an ArrayList, ArrayCollection, or XMLListCollection object, to the control's `dataProvider` property. For more information on using data providers, see "Data providers and collections" on page 898.

Data items can be simple data items such as String and Number objects, and more complicated data items such as Object and XMLNode objects. The following example shows a List control that displays data items represented by String data:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:List>
        <s:dataProvider>
            <mx:ArrayCollection>
                <fx:String>Flash</fx:String>
                <fx:String>Director</fx:String>
                <fx:String>Dreamweaver</fx:String>
                <fx:String>ColdFusion</fx:String>
            </mx:ArrayCollection>
         </s:dataProvider>
    </s:List>
</s:Application>
```

For simple data items, such as Strings and Numbers, the controls display the entire data item as a String. For more complicated data items, such as Objects, you can control the text displayed in the control based on the data item. For more information, see "Controlling the text displayed for each data item" on page 516.

Because `dataProvider` is the default property of the list-based controls, you do not have to specify a `<s:dataProvider>` child tag. Therefore, you can write the example as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSimpleNoDP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:List>
        <mx:ArrayCollection>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:List>
</s:Application>
```

Controlling the text displayed for each data item

The following example shows a List control where each data item is represented by an Object with three fields:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSimpleObjects.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:List id="myList"
        labelField="firstName">
        <mx:ArrayCollection>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayCollection>
    </s:List>
</s:Application>
```

When the data items contains multiple fields, you must configure the control to display the correct String. In the previous example, you use the `labelField` property to specify to display the firstName field in the control.

You can use several different mechanisms for controlling the String displayed by the control:

- Use the `labelFunction` property

  The `labelFunction` property lets you define a function that returns a String for display by the control. By default, the `labelFunction` property takes precedence over the `labelField` property.

- Use the `labelField` property

  The `labelField` property specifies the field of the data item to display in the control.

- Override the `itemToLabel()` method

  Override the `itemToLabel()` method to convert the data item to a String that is then passed to an item renderer. This method is defined on the SkinnableDataContainer class.

By default, `itemToLabel()` prioritizes `labelFunction` over `labelField`. If you override `itemToLabel()`, you can implement it however you like. Your implementation can then choose to use or ignore `labelField` and `labelFunction`. For an example that overrides this method, see "Passing data using the IItemRenderer.label property" on page 476.

When you use the `labelFunction` property, you define a function that calculates the String to display. Typically, the String is derived from the fields of the data item. The label function must have the following signature:

```
labelFunction(item:Object):String
```

The *`item`* parameter contains the data item. The function must return a String.

The following example uses a label function to concatenate the firstName and lastName fields of each data item for display by the control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSimpleObjectsLF.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function myLabelFunc(item:Object):String {
                return item.firstName + " " + item.lastName;
            }
        ]]>
    </fx:Script>
    <s:List id="myList"
        labelFunction="myLabelFunc">
        <mx:ArrayCollection>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayCollection>
    </s:List>
</s:Application>
```

Modifying the children of a list-based control

A list-based control creates list items based on the value of its `dataProvider` property. Typically, the control creates one list item for each data item. For example, the List control creates one list item for each data item, and the ButtonBar control creates one button for each data item.

Use the `addItem()` and `removeItem()` methods to manipulate the `dataProvider` property. A ButtonBar control automatically adds or removes children based on changes to the `dataProvider` property, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkBBarSimpleModDP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            // Add data item, and button, to the ButtonBar.
            private function addFlexToDP():void {
                myBB.dataProvider.addItem("Flex");
                addFlex.enabled=false;
                removeFlex.enabled=true;
            }
            // Remove data item, and button, from the ButtonBar.
            private function removeFlexToDP():void {
                myBB.dataProvider.removeItemAt(4);
                addFlex.enabled=true;
                removeFlex.enabled=false;
            }
        ]]>
    </fx:Script>
    <s:ButtonBar id="myBB">
        <mx:ArrayCollection>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:ButtonBar>

    <s:Button id="addFlex" label="Add Flex"
        click="addFlexToDP();"/>
    <s:Button id="removeFlex" label="Remove Flex"
        enabled="false"
        click="removeFlexToDP();"/>
</s:Application>
```

Setting the layout

The list-based controls support the `layout` property to let you control the layout of the data items. For example, the default layout of the List control is VerticalLayout. You can use any of the Flex layout classes with the list-based controls, and any custom layouts that you create.

***Note:** The Spark list-based controls do not support the BasicLayout class. Do not use BasicLayout with the Spark list-based controls.*

The following example overrides that layout to use TileLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListLayout.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:List width="200">
        <s:layout>
            <s:TileLayout/>
        </s:layout>
        <mx:ArrayCollection>
            <fx:String>Flex</fx:String>
            <fx:String>Flash Builder</fx:String>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
            <fx:String>Illustrator</fx:String>
        </mx:ArrayCollection>
    </s:List>
</s:Application>
```

For more information on Spark layouts, see “About Spark layouts” on page 420.

Working with list events

The list-based classes all support the following events:

- `caretChange` Dispatched after the focus has changed from one data item to another. For more information, see “Working with the caret item” on page 521.
- `change` Dispatched after the currently selected data item changes to another data item. This event is dispatched when the user interacts with the control.

  When you change the value of the `selectedIndex` or `selectedItem` properties programmatically, the control does not dispatch the `change` event. It dispatches the `valueCommit` event instead.

  For more information, see “Working with the selected item” on page 519.
- `changing` Dispatched before the currently selected data item changes. This event is dispatched when the user interacts with the control.

  Calling the `preventDefault()` method in the event handler prevents the selection from changing. This event is useful when you want to perform processing on the currently selected item before selection moves to a new data item, or to prevent an item from being selected.

  When you change the value of the `selectedIndex` or `selectedItem` properties programmatically, it does not dispatch the `changing` event. It dispatches the `valueCommit` event instead.

The event object for all events is spark.events.IndexChangeEvent.

Working with the selected item

The ListBase class adds support for item selection in the list of data items, and for controlling the text displayed in the list. Several properties of the ListBase class handle the selection of a data item in the control. These properties include the following:

- `requireSelection` Specifies that a data item must always be selected. The default is `false`. If `true`, then the first data item is selected by default until the user selects a list item, or until you set the `selectedIndex` property.

- `selectedIndex` The index of the currently selected data item. The index of items in the list-based controls is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items. If `requireSelection` is `false`, you can also set `selectedIndex` to -1 to deselect all data items.
- `selectedItem` The object from the data provider corresponding to the currently selected data item.

You often use these properties in event handlers, as the following example shows for the `change` event. In this example, the List control displays the firstName field of each data item. When you select a data item in the control, the event handler for the `change` event displays the index and the lastName field of the data item in TextArea controls.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSelected.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="450">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import spark.events.IndexChangeEvent;
            protected function selectionChangedHandler(event:IndexChangeEvent):void
            {
                var currentIndx:int = event.currentTarget.selectedIndex;
                var currentDataItem:Object = event.currentTarget.selectedItem;
                selIndex.text = String(currentIndx);
                selLName.text = String(currentDataItem.lastName);
            }
        ]]>
    </fx:Script>

    <s:List id="myList"
        labelField="firstName"
        change="selectionChangedHandler(event)">
        <mx:ArrayCollection>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayCollection>
    </s:List>

    <s:Label text="Selected index:"/>
    <s:TextArea id="selIndex" height="50"/>
    <s:Label text="Selected Last Name:"/>
    <s:TextArea id="selLName" height="50"/>

</s:Application>
```

The `currentTarget` property of the object passed to the event handler contains a reference to the List control. Therefore, the `currentTarget.selectedIndex` field contains the index of the selected item, and the `currentTarget.selectedItem` field contains a copy of the selected item.

Working with the caret item

The selected item is the item in the control selected by clicking on the item, or by using the arrow keys. For more information about determining the selected item, see "Working with the selected item" on page 519.

The caret item is the data item that currently has focus. The caret item can be the currently selected data item, or it can be a different item.

You use the `caretIndex` property to determine the index of the data item that currently has focus. The index is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items.

The following example displays the `selectedIndex` and the `caretIndex` properties for a List control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListCaret.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="5" paddingLeft="5"/>
    </s:layout>
    <s:Label text="The selected item is: {myList.selectedIndex}" />
    <s:Label text="The caret item is: {myList.caretIndex}" />
    <s:List id="myList">
        <s:dataProvider>
            <s:ArrayCollection>
                <fx:String>One</fx:String>
                <fx:String>Two</fx:String>
                <fx:String>Three</fx:String>
                <fx:String>Four</fx:String>
                <fx:String>Five</fx:String>
                <fx:String>Six</fx:String>
                <fx:String>Seven</fx:String>
                <fx:String>Eight</fx:String>
                <fx:String>Nine</fx:String>
                <fx:String>Ten</fx:String>
            </s:ArrayCollection>
        </s:dataProvider>
    </s:List>
</s:Application>
```

To see the difference between the `selectedIndex` and the `caretIndex` properties:

1 Select an item in the List control by clicking on it. Notice that because the item is both selected and in focus that the `selectedIndex` and the `caretIndex` properties have the same value.

2 Use the Up Arrow and Down Arrow keys to move the selected item. Notice that because the item is both selected and in focus that the `selectedIndex` and the `caretIndex` properties have the same value.

3 With an item selected:

- (On Windows) Hold down the Control key, and then use the Up Arrow and Down Arrow keys to move the caret index.
- (On OSX) Hold down the Command key, and then use the Up Arrow and Down Arrow keys to move the caret index.

Notice that the List control displays a blue box around the data item at the caret index, but the index of the selected item does not change. The data item at the caret index has focus, but is not selected. Press the Space key to select the data item at the caret index.

The list-based controls dispatch the `caretChange` event when the caret item changes.

Working with item renderers

Because the list-based classes are subclasses of the SkinnableDataContainer class, they require an item renderer to display the data items. The SkinnableDataContainer class does not define a default item renderer. However, the list-based classes all define a default item render as the following table shows:

Control	**Default item renderer**
ButtonBar	ButtonBarButton
ComboBox	DefaultItemRenderer
DropDownList	DefaultItemRenderer
List	DefaultItemRenderer
TabBar	ButtonBarButton with TabBarButtonSkin as the skin

You can override the default item renderer. For more information, see "Using an item renderer function with a Spark container" on page 481.

Using virtualization with list-based controls

A list-based control can represent any number of children. However, each child requires an instance of an item renderer. If the container has many children, you can notice performance degradation as you add more children to the container.

Instead of creating an item renderer for each child, you can use the `useVirtualLayout` property to configure the control to use a virtual layout. With virtual layout enabled, the control reuses item renderers so that it only creates item renderers for the currently visible children of the container. As a child is moved off the screen, possible by scrolling the container, a new child being scrolled onto the screen can reuse its item renderer.

The layout class associated with a component determines if the component supports virtual layout. Typically, you use the component's skin class to specify the layout class. By default, the skins for the Spark list-based classes have the `useVirtualLayout` property to `true` to enable virtual layout.

However, the default skins of some components use layouts that do not support the `useVirtualLayout` property. The TabBar and ButtonBar components use the ButtonBarHorizontalLayout which does not support the `useVirtualLayout` property or virtual layout.

For more information on virtualization, see "Using virtualization with Spark DataGroup and SkinnableDataContainer" on page 469.

Supporting view states in an item renderer for a list-based control

The Spark list-based controls support view states that you can use in a custom item renderer. Support for these view states is not required. If the host component attempts to set the item renderer to a view state that is not defined in the renderer, then the item renderer uses the normal view state.

Flex defines the following view states for item renderers.

- normal   The data item has no user interaction.
- hovered   The mouse is over the data item.
- selected   The data item is selected.
- dragging   The data item is being dragged.

- normalAndShowCaret The data item is in the normal state, and it has focus in the item list.
- hoveredAndShowCaret The data item is in the hovered state, and it has focus in the item list.
- selectedAndShowCaret The data item is in the selected state, and it has focus in the item list.

The default item renderers, including DefaultItemRenderer and DefaultComplexItemRenderer, support all of these view states.

The list-based controls add the ability to select a data item. Therefore, one of the view states corresponds to the data item being selected. The item renderer can use the selected view state to modify the appearance of the item when selected.

Several view states are associated with the caret index. The caret index identifies the data item that has focus. That item does not have to be the selected item,

The following example shows a simple item renderer for the List class that supports the selected view state. In this example, the item renderer shows the selected item in red with a black border:

```
<?xml version="1.0" encoding="utf-8"?>
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[

            override public function set label(value:String):void {
                super.label = value;
                labelDisplay.text = label;
            }
        ]]>
    </fx:Script>
    <s:states>
        <s:State name="normal"/>
        <s:State name="hovered"/>
        <s:State name="selected"/>
    </s:states>

    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke.selected>
            <s:SolidColorStroke
                color="0x000000"
                weight="1"/>
        </s:stroke.selected>
        <s:fill>
            <s:SolidColor
                color.normal="0xFFFFFF"
                color.hovered="0xCEDBEF"
                color.selected="0x00FF00"/>
        </s:fill>
    </s:Rect>
    <s:Label id="labelDisplay"
        fontWeight.selected="bold"
        verticalCenter="0"
        left="3" right="3"
        top="6" bottom="4"/>
</s:ItemRenderer>
```

This item renderer does not define all possible view states. If the host component attempts to set it to any undefined view state, the item renderer uses the normal state.

The following application uses this item renderer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListIR.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:List itemRenderer="myComponents.ListSelectedIR">
        <s:dataProvider>
            <mx:ArrayCollection>
                <fx:String>Flash</fx:String>
                <fx:String>Director</fx:String>
                <fx:String>Dreamweaver</fx:String>
                <fx:String>ColdFusion</fx:String>
            </mx:ArrayCollection>
         </s:dataProvider>
    </s:List>
</s:Application>
```

Using a mobile item renderer with a Spark list-based control

Flex provides two item renderers that you can use with Spark list-based controls in mobile applications. These item renderers, spark.components.LabelItemRenderer and spark.components.IconItemRenderer, are written in ActionScript to provide the highest level of performance required by mobile applications.

The default theme for mobile applications built in Flex is the Mobile theme. When you use the Mobile theme, Flex automatically uses the LabelItemRenderer for the Flex list-based controls. Like the DefaultItemRenderer, the LabelItemRenderer displays a single text field for each data item.

You can use the IconItemRenderer in a mobile application. The IconItemRenderer displays two text fields, one above the other, and two icons for every data item.

The following example shows a List control defined in the view of a mobile application. In this example, the List control uses the IconItemRenderer to display the first name, last name, and an icon for every data item in the list:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\mobile\views\ContactsMobileIconIR.mxml -->
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    title="Contacts View Main">
    <s:layout>
        <s:VerticalLayout paddingTop="10"/>
    </s:layout>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        .myFontStyle {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>

    <s:Label text="Select a contact name"/>
    <s:List id="myList"
        width="100%" height="100%"
        labelField="firstName">
        <s:itemRenderer>
            <fx:Component>
                <s:IconItemRenderer messageStyleName="myFontStyle" fontSize="25"
                    labelField="firstName"
                    messageField="lastName"
                    decorator="@Embed(source='../assets/logo_small.jpg')"/>
            </fx:Component>
        </s:itemRenderer>
        <s:ArrayCollection>
         <fx:Object firstName="Dave" lastName="Duncam" company="Adobe" phone="413-555-1212"/>
          <fx:Object firstName="Sally" lastName="Smith" company="Acme" phone="617-555-1491"/>
          <fx:Object firstName="Jim" lastName="Jackson" company="Beta" phone="413-555-2345"/>
          <fx:Object firstName="Mary" lastName="Moore" company="Gamma" phone="617-555-1899"/>
        </s:ArrayCollection>
    </s:List>
</s:View>
```

Spark List control

The Spark List control displays a list of data items. Its functionality is very similar to that of the SELECT form element in HTML. The user can select one or more items from the list. The List control automatically displays horizontal and vertical scroll bar when the list items do not fit the size of the control.

The List control is a subclass of the SkinnableDataContainer container. Therefore, you can set the layout of the List control, apply a skin to it, and define a custom item renderer. For more information on the SkinnableDataContainer container, see “The Spark DataGroup and Spark SkinnableDataContainer containers” on page 466. To define a look and feel of items displayed in the List control, read more about “Define a custom Spark item renderer” on page 470.

Creating a Spark List control

You use the `<s:List>` tag to define a Spark List control. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the data for the List control by using the `dataProvider` property of the control. However, because `dataProvider` is the List control's default property, you do not have to specify a `<s:dataProvider>` child tag of the `<s:List>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSimpleNoDP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:List>
        <mx:ArrayCollection>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:List>
</s:Application>
```

The default skin for the List control, ListSkin, defines scroll bars by including a Scroller component in the skin class. You can reference the Scroller component from the List by using the `List.scroller` skin part. Therefore, if the data items do not fit in the size of the control, Flex automatically display scroll bars, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSimpleScroll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:List height="100">
        <s:dataProvider>
            <mx:ArrayCollection>
                <fx:String>Flex</fx:String>
                <fx:String>Flash Builder</fx:String>
                <fx:String>Flash</fx:String>
                <fx:String>Director</fx:String>
                <fx:String>Dreamweaver</fx:String>
                <fx:String>ColdFusion</fx:String>
                <fx:String>Illustrator</fx:String>
                <fx:String>PhotoShop</fx:String>
            </mx:ArrayCollection>
        </s:dataProvider>
    </s:List>
</s:Application>
```

Using the styles of the Spark List control

The List control defines several styles that you can use to modify the appearance of the control. You can use styles to make some visual changes to the control, but you create a skin for total control over the display.

The following example uses the `alternatingItemColors` and `rollOverColor` styles with the List control. The `alternatingItemColors` styles defines an Array with two entries, #66FFFF for light blue and #33CCCC for a blue-gray. Therefore, the rows of the List control alternate between these two colors. If you specify a three-color array, the rows alternate among the three colors, and so on.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\spark\SparkListSimpleStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:List
        alternatingItemColors="[#66FFFF, #33CCCC]"
        rollOverColor="#CC6600">
        <s:dataProvider>
            <mx:ArrayCollection>
                <fx:String>Flex</fx:String>
                <fx:String>Flash Builder</fx:String>
                <fx:String>Flash</fx:String>
                <fx:String>Director</fx:String>
                <fx:String>Dreamweaver</fx:String>
                <fx:String>ColdFusion</fx:String>
            </mx:ArrayCollection>
         </s:dataProvider>
    </s:List>
</s:Application>
```

Handling multiple selection in the Spark List control

The List control lets you select multiple items in the control. To enable multiple selection, set the `allowMultipleSelection` property to `true`. The default value is `false`.

After enabling multiple selection, select the first data item in the List by clicking on it. Then hold down the Control key to select additional items. If the item is currently deselected, clicking on it while holding down the Control key selects it. If the item is already selected, clicking on it while holding down the Control key deselects it.

You can also use the Shift key to select a range of data items. To select a range, select the first data item in the List by clicking on it. Then hold down the Shift key to select one additional item. The List control selects all data items between the two data items.

The List control defines two Vector properties that you use with multiple selection: `selectedIndices` and `selectedItems`. The `selectedIndices` Vector is of type int, and the `selectedItems` Vector is of type Object. These Vectors contain a list of the selected indices and selected data items in the reverse order in which they were selected. That means the first element in each Vector corresponds to the last item selected. The length of each Vector is the same as the number of selected items in the control. If your list contains 100 data items, these Vectors can be up to 100 elements long.

The following example enables multiple selection for the List control. It displays the index of all selected items in one TextArea control, and the data item itself in a second TextArea control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkListSimpleMultipleSelection.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="450">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import spark.events.IndexChangeEvent;

            private function myChangedHandler(event:IndexChangeEvent):void {

                var selIndices:Vector.<int> = event.currentTarget.selectedIndices;
                var selItems:Vector.<Object> = event.currentTarget.selectedItems;
                var numItems:Number = selIndices.length;

                selIndicesTA.text = "";
                selItemsTA.text = "";

                for (var i:Number = 0; i<numItems; i++)
                {
                    selIndicesTA.text = selIndicesTA.text + selIndices[i] + "\n";
                    selItemsTA.text = selItemsTA.text + selItems[i] + "\n";
                }
            }
        ]]>
    </fx:Script>
    <s:List allowMultipleSelection="true"
        change="myChangedHandler(event);">
        <mx:ArrayCollection>
            <fx:String>Flex</fx:String>
            <fx:String>Flash Builder</fx:String>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:List>

    <s:Label text="Selected indices"/>
    <s:TextArea id="selIndicesTA" height="75"/>
    <s:Label text="Selected items"/>
    <s:TextArea id="selItemsTA" height="75"/>
</s:Application>
```

Selecting multiple items does not dispatch the `changing` event. It only dispatches the `change` event.

Spark List control user interaction

The user clicks individual data items to select them, and holds down the Control and Shift keys while clicking to select multiple items. You must set the `allowMultipleSelection` property to `true` to allow multiple selection.

A List control shows the number of data items that fit in the display. Paging down through the data items displayed by a 10-line List control shows records 0-9, 10-19, 20-29, and so on.

The List control has the following default keyboard navigation features:

Key	Action
Up Arrow	Moves selection up one item. Hold down the Control key (Windows) or Command key (OSX) to only move focus, but not change the selected item.
Down Arrow	Moves selection down one item. Hold down the Control key (Windows) or Command key (OSX) to only move focus, but not change the selected item.
Left Arrow	Moves selection one column to the left, if you are using HorizontalLayout or TileLayout only. Hold down the Control key (Windows) or Command key (OSX) to only move focus, but not change the selected item.
Right Arrow	Moves selection one column to the right, if you are using HorizontalLayout or TileLayout only. Hold down the Control key (Windows) or Command key (OSX) to only move focus, but not change the selected item.
Page Up	Moves selection up one page.
Page Down	Moves selection down one page.
Home	Moves selection to the top of the list.
End	Moves selection to the bottom of the list.
Alphanumeric keys	Jumps to the next item with a label that begins with the character typed.
Control	Toggle key. Allows for multiple (noncontiguous) selection and deselection. Works with key presses, click selection, and drag selection.
Shift	Contiguous selection key. Allows for contiguous selections. Works with key presses, click selection, and drag selection.

Spark ButtonBar and TabBar controls

The Spark ButtonBar control defines a horizontal row of related buttons with a common appearance. The controls define a single event, the `itemClick` event, that is dispatched when any button in the control is selected.

The buttons maintain their state, either selected or deselected. Only one button in the ButtonBar control can be in the selected state. That means when you select a button in a ButtonBar control, the button stays in the selected state until you select a different button.

The TabBar control is similar to the ButtonBar control, except that is displays a horizontal row of tabs instead of buttons.

Adobe Engineer Joan Lafferty creates a custom skin for the Spark ButtonBar control in Adding toolTips to a Spark ButtonBar.

Creating a Spark ButtonBar control

You create a Spark ButtonBar control in MXML by using the `<s:ButtonBar>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the data for the ButtonBar control by using the `dataProvider` property of the control. However, because `dataProvider` is the ButtonBar control's default property, you do not have to specify a `<s:dataProvider>` child tag of the `<s:ButtonBar>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkBBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingLeft="20" paddingTop="20"/>
    </s:layout>
    <s:ButtonBar>
        <mx:ArrayCollection>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:ButtonBar>
</s:Application>
```

This example creates a row of four Button controls.

Creating a Spark TabBar control

You create a Spark TabBar control in MXML by using the `<s:TabBar>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the data for the TabBar control by using the `dataProvider` property of the control. However, because `dataProvider` is the TabBar control's default property, you do not have to specify a `<s:dataProvider>` child tag of the `<s:TabBar>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkTabBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingLeft="20" paddingTop="20"/>
    </s:layout>
    <s:TabBar>
        <mx:ArrayCollection>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:TabBar>
</s:Application>
```

This example creates a row of four tabs.

Using the Spark ButtonBar and TabBar controls with an MX ViewStack container

You can use the Spark ButtonBar and TabBar controls to set the active child of a ViewStack container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\navigators\VSSparkButtonBar.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:VGroup>
        <!-- Create a Spark ButtonBar control to navigate
            the ViewStack container. -->
        <s:ButtonBar dataProvider="{myViewStack}"/>
        <!-- Define the ViewStack and the three child containers. -->
        <mx:ViewStack id="myViewStack"
            borderStyle="solid"
            width="100%">

            <s:NavigatorContent id="search" label="Search">
                <s:Label text="Search Screen"/>
            </s:NavigatorContent>
            <s:NavigatorContent id="custInfo" label="Customer Info">
                <s:Label text="Customer Info"/>
            </s:NavigatorContent>
            <s:NavigatorContent id="accountInfo" label="Account Info">
                <s:Label text="Account Info"/>
            </s:NavigatorContent>
        </mx:ViewStack>
    </s:VGroup>
</s:Application>
```

For more information on the ViewStack control, see the “MX ViewStack navigator container” on page 630.

Using skins and styles with the Spark ButtonBar and TabBar controls

The default skin class for the ButtonBar control, spark.skins.spark.ButtonBarSkin, references three skin classes to define the buttons:

- spark.skins.spark.ButtonBarFirstButtonSkin The skin for the first button.
- spark.skins.spark.ButtonBarMiddleButtonSkin The skin for the middle buttons.
- spark.skins.spark.ButtonBarLastButtonSkin The skin for the last button.

The individual button skins define the buttons to be instances of the ButtonBarButton control. You can define custom skin classes for the ButtonBar control. Your custom skin classes can then use any control to represent each data item, not just the ButtonBarButton control.

The tabs on a TabBar control are all drawn in the same way. Therefore, the default skin class for the TabBar control, spark.skins.spark.TabBarSkin, references only a single skin class to draw the tabs: spark.skins.spark.TabBarButtonSkin.

You can also set styles to control the appearance of the buttons and tabs. The following example uses styles on the ButtonBar and ButtonBarButton controls to define the appearance of the individual buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkBBStyled.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingLeft="20" paddingTop="20"/>
    </s:layout>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|ButtonBar s|ButtonBarButton:upAndSelected,
        s|ButtonBar s|ButtonBarButton:overAndSelected,
        s|ButtonBar s|ButtonBarButton:downAndSelected,
        s|ButtonBar s|ButtonBarButton:disabledAndSelected {
            chromeColor: #663366;
            color: #9999CC;
        }
        s|ButtonBar {
            chromeColor: #9999CC;
            color: #663366;
        }
    </fx:Style>

    <s:ButtonBar
        requireSelection="true"
        width="320">
        <mx:ArrayCollection>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:ButtonBar>
</s:Application>
```

In this example, you use CSS descendant selectors to style the ButtonBarButton controls of the ButtonBar, and to set styles directly on the ButtonBar control. The styles of the ButtonBarButton controls define the appearance of the controls for different view states of the ButtonBarButton control.

For more information on using descendant selectors, see “Using Cascading Style Sheets” on page 1504.

Handling Spark ButtonBar and TabBar events

The ButtonBar and TabBar controls dispatches a `change` event when you select a button or tab. The event object passed to the event listener is of type IndexChangeEvent. Access properties of the event object to determine the index of the selected button or tab, and other information. The index of the first button or tab is 0. For more information about the event object, see the description of the ItemClickEvent class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The ButtonBar control in the following example defines an event listener, named `changeHandler()`, for the `change` event:

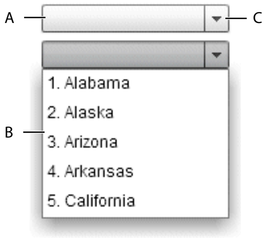
```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkBBarEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import spark.events.IndexChangeEvent;

            private function changeHandler(event:IndexChangeEvent):void {
                myTA.text="Selected button index: " +
                String(event.newIndex) + "\n" +
                "Selected button label: " +
                event.target.selectedItem;
            }
        ]]>
    </fx:Script>

    <s:ButtonBar
        change="changeHandler(event);">
        <mx:ArrayCollection>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:ArrayCollection>
    </s:ButtonBar>

    <s:TextArea id="myTA" width="250" height="100"/>
</s:Application>
```

In this example, the click handler displays the index and label of the selected button in a TextArea control in response to an `change` event.

Spark DropDownList control

The DropDownList control pops up a list when the user selects the anchor. When the user selects an item from the drop-down list, the data item appears in the prompt area of the control.

The following image shows the DropDownList control:



***A.** Prompt in a closed DropDownList* ***B.** Drop-down list in an open DropDownList* ***C.** Anchor in a closed DropDownList*

Unlike the ComboBox control, the DropDownList control is not editable.

Peter DeHaan has several examples of using the Spark DropDownList on flexexamples.com.

StackOverflow shows how to adjust the height of an open DropDownList with a custom skin, and how to bind the data field to XML.

When the drop-down list is closed, clicking the anchor button opens it.

When the drop-down list is open:

- Clicking the anchor button closes the drop-down list and commits the currently selected data item.
- Clicking outside the drop-down list closes the drop-down list and commits the currently selected data item.
- Clicking on a data item selects that item and closes the drop-down list.
- If the `requireSelection` property is `false`, clicking on a data item while pressing the Control key deselects the item and closes the drop-down list.

For a complete list of keyboard command for the DropDownList control, see “Spark DropDownList control user interaction” on page 536.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Create a Spark DropDownList control

You define a DropDownList control in MXML by using the `<s:DropDownList>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block.

You specify the data for the DropDownList control by using the `dataProvider` property. However, because `dataProvider` is the DropDownList control’s default property, you do not have to specify a `<s:dataProvider>` child tag of the `<s:DropDownList>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDDSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:DropDownList width="140">
        <mx:ArrayCollection>
            <fx:String>Alabama</fx:String>
            <fx:String>Alaska</fx:String>
            <fx:String>Arizona</fx:String>
            <fx:String>Arkansas</fx:String>
            <fx:String>California</fx:String>
        </mx:ArrayCollection>
    </s:DropDownList>
</s:Application>
```

Set the prompt in the Spark DropDownList control

By default, the prompt area is empty when the control first loads. Selecting a data item in the drop-down list sets the prompt to the data item.

Use the `prompt` property to define the text to display when the DropDownList loads, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDDSimplePrompt.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingLeft="10"
            paddingTop="10"/>
    </s:layout>
    <s:DropDownList width="140"
        prompt="Select a state">
        <mx:ArrayCollection>
            <fx:String>Alabama</fx:String>
            <fx:String>Alaska</fx:String>
            <fx:String>Arizona</fx:String>
            <fx:String>Arkansas</fx:String>
            <fx:String>California</fx:String>
        </mx:ArrayCollection>
    </s:DropDownList>
</s:Application>
```

If you set the `requireSelection` property to `true`, the control ignores the `prompt` property. Instead, it sets the prompt area and the `selectedIndex` property to the first data item in the control.

Alternatively, you can set the `selectedIndex` property to any data item. The prompt area displays the data item specified by the `selectedIndex` property. In the following example, you set the `selectedIndex` property to 4 to select California by default when the application starts:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDDSimpleSetSelectedIndex.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingLeft="10"
            paddingTop="10"/>
    </s:layout>
    <s:DropDownList width="140"
        selectedIndex="4">
        <mx:ArrayCollection>
            <fx:String>Alabama</fx:String>
            <fx:String>Alaska</fx:String>
            <fx:String>Arizona</fx:String>
            <fx:String>Arkansas</fx:String>
            <fx:String>California</fx:String>
        </mx:ArrayCollection>
    </s:DropDownList>
</s:Application>
```

Handle Spark DropDownList events

The DropDownList control inherits the events of the ListBase class, and adds the following events:

- `close` Dispatched when the drop-down list closes for any reason.
- `open` Dispatched when the drop-down list opens for any reason.

Spark DropDownList control user interaction

A page in the DropDownList control is defined as the number of data items that fit in the drop-down list. Paging down through the data displayed by a six-line control shows records 0-5, 6-11, 12-17, and so on.

Keyboard navigation when DropDownList is closed

The DropDownList control has the following keyboard navigation features when the drop-down list is closed:

Key	Action
Alphanumeric keys	Displays the closest match in the item list. The DropDownList does not open.
Up Arrow	Moves selection up one item.
Down Arrow	Moves selection down one item.
Control + Down Arrow	Opens the drop-down list.
Page Up	Moves the selection up one page. If the `selectedIndex` is currently -1, do nothing.
Page Down	Moves the selection down one page. If you are at the end of the list, do nothing.
Home	Moves selection to the top of the drop-down list.
End	Moves selection to the bottom of the drop-down list.

Keyboard navigation when DropDownList is open

The DropDownList control has the following keyboard navigation features when the drop-down list is open:

Key	Action
Alphanumeric keys	Scrolls to and highlights the closest match in the item list.
Up Arrow	Moves selection up one item but does not commit the selection until the drop-down list closes.
Down Arrow	Moves selection down one item but does not commit the selection until the drop-down list closes.
Control + Up Arrow	Closes the drop-down list and commits the selection.
Escape	Closes the drop-down list but do not commit the selection.
Page Up	Moves the selection to the top of the data items currently displayed but does not commit the selection until the drop-down list closes.
Page Down	Moves the selection to the bottom of the data items currently displayed but does not commit the selection until the drop-down list closes.
Home	If `selectedIndex` = -1, do not change the currently selected data item. Otherwise, moves selection to the first data item in the drop-down list. Does not commit the selection until the drop-down list closes.
End	Moves selection to the last data item in the drop-down list. Does not commit the selection until the drop-down list closes.

Spark ComboBox control

The ComboBox control is a child class of the DropDownListBase control. Like the DropDownList control, when the user selects an item from the drop-down list in the ComboBox control, the data item appears in the prompt area of the control.

Differences between ComboBox and DropDownList

One difference between the controls is that the prompt area of the ComboBox control is implemented by using the TextInput control, instead of the Label control for the DropDownList control. Therefore, a user can edit the prompt area of the ComboBox control to enter a value that is not one of the predefined options; the values in the DropDownList cannot be edited.

For example, the DropDownList control only lets the user select from a list of predefined items in the control. The ComboBox control lets the user either select a predefined item, or enter a new item into the prompt area. When the user enters a new item, the `selectedIndex` property is set to -3. Your application can recognize that a new item has been entered and, optionally, add it to the list of items in the control.

Searching

The ComboBox control also searches the item list as the user enters characters into the prompt area. As the user enters characters, the drop-down area of the control opens. It then and scrolls to and highlights the closest match in the item list.

Create a Spark ComboBox control

You define a ComboBox control in MXML by using the `<s:ComboBox>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block.

You specify the data for the ComboBox control by using the `dataProvider` property. However, because `dataProvider` is the ComboBox control's default property, you do not have to specify a `<s:dataProvider>` child tag of the `<s:ComboBox>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkCBSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="5" paddingLeft="5"/>
    </s:layout>
    <s:Label text="The selected index is: {myCB.selectedIndex}"/>
    <s:Label text="The selected item is: {myCB.selectedItem}"/>
    <s:ComboBox id="myCB" width="140" prompt="Select a Color">
        <s:dataProvider>
            <mx:ArrayList>
                <fx:String>Red</fx:String>
                <fx:String>Orange</fx:String>
                <fx:String>Yellow</fx:String>
                <fx:String>Blue</fx:String>
                <fx:String>Green</fx:String>
            </mx:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

This example demonstrates the following functionality of the ComboBox control:

- When you select an item in the drop-don list, the item appears in the prompt area of the control. The `ComboBox.selectedIndex` property contains the index of the selected item in the data provider of the control. The `ComboBox.selectedItem` property contains the selected data item.
- As you enter characters in the prompt area, it scrolls to and highlights the closest match in the item list.

You can control the matching algorithm by creating a custom callback function. You then specify that function as the value of the `ComboBox.itemMatchingFunction` property. The callback function returns a Vector.<int> that contains the index of each item in the data provider that matches the input. To disable matching, create a callback function that returns an empty Vector.<int>.

- If you enter a value in the prompt area that does not match an item in the list, and commit the item, the index of the selected item is -3. This value is determined by the `ComboBox.CUSTOM_SELECTED_ITEM` constant.

  Commit a new item by pressing the Enter or Tab key while the cursor is in the prompt area.

Handle events for the Spark ComboBox control

You commonly use the following events with the ComboBox control:

- The ComboBox control dispatches the `changing` event after each character is entered in the prompt area.
- The ComboBox control dispatches the `change` event when the user selects an existing item in the drop-down list, or enters a new item in the prompt area.

You can also add event listeners to the `textInput` skin part of the ComboBox control to handle events on the TextInput control. For example, the following line adds an event handler for the `change` event to the `textInput` skin part of the ComboBox named myCB:

```
myCB.textInput.addEventListener('change', myChangeHandler);
```

Process new data items entered into the Spark ComboBox control

When using the ComboBox control, you must decide on how your application handles new values entered into the prompt are of the control. For example, your application can recognize the new value and perform some action, it can provide user feedback with a list of allowed values, and it can add the item to the current list of data items.

The following actions occur when the user enters a value into the prompt area of the control, and the value is not found in the current list of data items:

- The ComboBox control dispatches the `change` event when the user commits the change, typically by pressing the Enter or Tab key.
- The `ComboBox.selectedIndex` property is set to `ComboBox.CUSTOM_SELECTED_ITEM`. The default value of this constant is -3.
- The ComboBox control calls the function defined by the `ComboBox.labelToItemFunction` property to convert the new value to the same data type as the data items in the data provider.

  If your data items are simple data types, such as String or int, then you can use the default function defined by the `labelToItemFunction` property. If your data items are Objects, then you typically have to write your own function.
- The converted value returned by the function specified by the `ComboBox.labelToItemFunction` property is written to the `ComboBox.selectedItem` property.

The following example uses the `change` event to recognize that a new item was entered into the prompt, and adds it to the data provider of the control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkCBAddItem.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="5" paddingLeft="5"/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import spark.events.IndexChangeEvent;

            // Event handler to determine if the selected item is new.
            protected function myCB_changeHandler(event:IndexChangeEvent):void
            {
                // Determine if the index specifies a new data item.
                if(myCB.selectedIndex == spark.components.ComboBox.CUSTOM_SELECTED_ITEM)
                    // Add the new item to the data provider.
                    myCB.dataProvider.addItem(myCB.selectedItem);
            }
        ]]>
    </fx:Script>

    <s:Label text="The selected index is: {myCB.selectedIndex}"/>
    <s:Label text="The selected item is: {myCB.selectedItem}"/>
    <s:ComboBox id="myCB" width="140" change="myCB_changeHandler(event);">
        <s:dataProvider>
            <mx:ArrayList>
                <fx:String>Red</fx:String>
                <fx:String>Orange</fx:String>
                <fx:String>Yellow</fx:String>
                <fx:String>Blue</fx:String>
                <fx:String>Green</fx:String>
            </mx:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

In this example, the ComboBox control initially defines five colors. The user can either select one of the predefined colors, or enter a new one in the prompt area. When the user commits the new item, by pressing the Enter or Tab key, the application uses the `change` even to add the new item to the data provider.

The ComboBox in the previous example used simple String data for its data items. That means the application can use the default function defined by the `ComboBox.labelToItemFunction` property to convert new values to the same data type as the data items in the data provider.

However, your ComboBox control can use more complicated data items. In that case, you write a custom function to convert new data items to the format required by the data provider, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkCBAddItemObj.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="5" paddingLeft="5"/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import spark.events.IndexChangeEvent;

            // Define a custom function for the labelFunction property
            // to display an Object in the ComboBox control.
            public function myLabelFunc(item:Object):String {
                return item.firstName + " " + item.lastName;
            }

            // Define a custom function for the labelToItemFunction property
            // to convert the new value to an Object of the correct format
            // for storage in the data provider of the control.
            public function myLabelToItemFunc(value:String):Object {
                var tempObj:Object = new Object();
                var spaceChar:int = value.indexOf(' ');
                tempObj.firstName = value.substr(0, spaceChar);
                tempObj.lastName = value.substr(spaceChar+1, value.length);
                return tempObj;
            }

            // Event handler to determine if the selected item is new.
            protected function myCB_changeHandler(event:IndexChangeEvent):void
            {
                // Determine if the index specifies a new data item.
                if(myCB.selectedIndex == spark.components.ComboBox.CUSTOM_SELECTED_ITEM)
                    // Add the new item to the data provider.
                    myCB.dataProvider.addItem(myCB.selectedItem);
            }
        ]]>
    </fx:Script>

    <s:Label text="The selected index is: {myCB.selectedIndex}"/>
    <s:Label text="The selected item is: {myCB.selectedItem.firstName + ' '
             + myCB.selectedItem.lastName}"/>
    <s:ComboBox id="myCB" width="140"
        labelFunction="myLabelFunc"
        labelToItemFunction="myLabelToItemFunc"
        change="myCB_changeHandler(event);">
        <s:dataProvider>
            <mx:ArrayList>
                <fx:Object firstName="Steve" lastName="Smith"/>
                <fx:Object firstName="John" lastName="Jones"/>
                <fx:Object firstName="Mary" lastName="Moore"/>
            </mx:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

In this example, the data items are represented by Objects. Each Object contains two fields: `firstName` and `lastName`.

This example adds two new functions. The first, `myLabelFunction()`, converts the Objects in the data provider of the control to a String for display by the ComboBox control. Use the `labelFunction` property to assign this function to the ComboBox control. For more information, see "Controlling the text displayed for each data item" on page 516.

The second function, `myLabelToItemFunc()`, converts a new data item into the format required by the data provider of the control. Use the `labelToItemFunction` property to assign this function to the ComboBox control. The ComboBox control writes the value returned by the function to the `selectedItem` property.

The signature of the function specified to the `labelToItemFunction` property is shown below:

```
function funcName(value:String):Object {}
```

The value passed to the function is the String entered into the prompt area of the control. The function returns an Object written to the `selectedItem` property.

Spark ComboBox control user interaction

A page in the ComboBox control is defined as the number of data items that fit in the drop-down list. Paging down through the data displayed by a six-line control shows records 0-5, 6-11, 12-17, and so on.

Keyboard navigation when ComboBox is closed

The ComboBox control has the following keyboard navigation features when the drop-down list is closed:

Key	Action
Alphanumeric keys	Opens the drop-down list. The drop-down list it scrolls to and highlights the closest match in the item list.
Enter	Commit the highlighted data item, if one is highlighted. If no data item is highlighted, commit the value of the prompt area.
Up Arrow	Moves selection up one item.
Down Arrow	Moves selection down one item.
Control + Down Arrow	Opens the drop-down list.
Page Up	Moves the selection up one page. If the `selectedIndex` is currently -1, do nothing.
Page Down	Moves the selection down one page. If you are at the end of the list, do nothing.
Home	Moves selection to the top of the drop-down list.
End	Moves selection to the bottom of the drop-down list.

Keyboard navigation when ComboBox is open

The ComboBox control has the following keyboard navigation features when the drop-down list is open:

Key	Action
Alphanumeric keys	Scrolls to and highlights the closest match in the item list.
Enter	Close the drop-down list. Commit the highlighted data item, if one is highlighted. If no data item is highlighted, commit the value of the prompt area.
Up Arrow	Moves selection up one item but does not commit the selection until the drop-down list closes.
Down Arrow	Moves selection down one item but does not commit the selection until the drop-down list closes.
Control + Up Arrow	Closes the drop-down list and commits the selection.

Key	Action
Escape	Closes the drop-down list but do not commit the selection.
Page Up	Moves the selection to the top of the data items currently displayed but does not commit the selection until the drop-down list closes.
Page Down	Moves the selection to the bottom of the data items currently displayed but does not commit the selection until the drop-down list closes.
Home	If `selectedIndex` = -1, do not change the currently selected data item. Otherwise, moves selection to the first data item in the drop-down list. Does not commit the selection until the drop-down list closes.
End	Moves selection to the last data item in the drop-down list. Does not commit the selection until the drop-down list closes.

Mouse navigation when ComboBox is closed

The ComboBox control has the following mouse navigation features when the drop-down list is closed:

Mouse action	Action
Click in the prompt area	Puts focus in the prompt area. If it was not previously in focus, then all of text in the prompt area is selected.
Click in open button	Opens the drop-down list, removes the focus ring from the ComboBox control, removes focus from the prompt area, and highlights the closest match to the input string.

Mouse navigation when ComboBox is open

The ComboBox control has the following mouse navigation features when the drop-down list is open:

Mouse action	Action
Click in the prompt area	Puts focus in the prompt area. If it was not previously in focus, then all of text in the prompt area is selected.
Click in open button	Closes the drop-down list and commit the selection. Removes focus from the ComboBox and control and from the prompt area.

Use prompt text with the Spark ComboBox control

The Spark ComboBox control supports adding prompt text. Prompt text appears in the prompt area of the control when no item is selected. Prompt text disappears when you focus on the control. When you remove focus without selecting an item, the prompt text reappears.

You set prompt text on a ComboBox control by using the `prompt` property. This property can be bound to the value of another property or it can be static text.

The following example displays prompt when the application first starts. When you focus on the ComboBox control, the prompt text disappears. If you click the Reset button, the value of the ComboBox's `selectedIndex` property is reset to -1, and the prompt text reappears.

```
<?xml version="1.0"?>
<!-- dpcontrols/sparkdpcontrols/SparkCBPrompt.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
        [Bindable]
        private var promptText:String = "Choose Game";

        /* When you set selectedIndex to -1 and remove focus, the prompt text reappears. */
        private function resetCB():void {
            myComboBox.selectedIndex = -1;
        }
      ]]>
  </fx:Script>

    <s:VGroup>
        <s:ComboBox id="myComboBox" prompt="{promptText}">
            <s:dataProvider>
                <s:ArrayList>
                    <fx:String>Durche die Wuste</fx:String>
                    <fx:String>Samurai</fx:String>
                    <fx:String>Traumfabrik</fx:String>
                </s:ArrayList>
            </s:dataProvider>
        </s:ComboBox>
        <s:Button label="Reset" click="resetCB()"/>
    </s:VGroup>
</s:Application>
```

The prompt text is defined by the TextInput subcontrol of the ComboBox control. It cannot include any HTML markup, but can be styled using CSS. To style the prompt text, use the `normalWithPrompt` (for the normal state) and `disabledWithPrompt` (for the disabled state) pseudo selectors on a TextInput or SkinnableTextBase class in your CSS, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/sparkdpcontrols/SparkCBPromptStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|TextInput:normalWithPrompt {
            color: #FF3366;
            fontStyle: italic;
        }
        s|TextInput:disabledWithPrompt {
            color: #CCFFFF;
            fontStyle: italic;
        }
    </fx:Style>
  <fx:Script>
    <![CDATA[
        [Bindable]
        private var promptText:String = "Choose Game";

        /* When you set selectedIndex to -1 and remove focus, the prompt text reappears. */
        private function resetCB():void {
            myComboBox.selectedIndex = -1;
        }
      ]]>
  </fx:Script>

    <s:VGroup>
        <s:ComboBox id="myComboBox" prompt="{promptText}">
            <s:dataProvider>
                <s:ArrayList>
                    <fx:String>Durche die Wuste</fx:String>
                    <fx:String>Samurai</fx:String>
                    <fx:String>Traumfabrik</fx:String>
                </s:ArrayList>
            </s:dataProvider>
        </s:ComboBox>
        <s:Button label="Reset" click="resetCB()"/>
    </s:VGroup>
</s:Application>
```

Additional information

The flepstudio.org blog shows how to load an external XML file into a ComboBox.

Flex4Fun has an example that adds icons to the ComboBox itemRender and another example that adds icons without creating a custom itemRenderer.

Sensaran’s Weblog has an example that adds an editable ComboBox to a DataGrid.

Spark DataGrid and Grid controls

The Spark DataGrid control displays a row of column headings above a scrollable grid. The grid consists of a collection of individual cells arranged in rows and columns. The DataGrid control is designed to support smooth scrolling through a large number of rows and columns.

The Spark DataGrid control is implemented as a skinnable wrapper around the Spark Grid control. The Grid control defines the columns of the data grid, and much of the functionality of the DataGrid control itself.

The DataGrid skin is responsible for laying out the grid, the column headers, and the scroller. The skin also configures the graphic elements used to render visual elements used as indicators, separators, and backgrounds. The DataGrid skin defines a default item renderer used to display the contents of each cell.

***Note:** Because the DataGrid control is implemented as a wrapper on the Grid control, they share much of the same functionality. Therefore, many of the examples below are for the DataGrid control, but can be easily modified to apply to the Grid control.*

The Spark DataGrid control provides the following features:

- Interactive column width resizing
- Control of column visibility
- Cell and row selection
- Single item and multiple item selection modes
- Customizable column headers
- Cell editing
- Column sorting
- Custom item renderers and item editors
- Custom skins to control all aspects of the appearance of the DataGrid control

Adobe Product Evangelist James Ward provides an overview of the Spark DataGrid control in the video Overview of Spark DataGrid in Flex.

Adobe Senior Computer Scientist Alex Harui creates a custom skin for the Spark DataGrid control in Spark DataGrid Footers, and shares his sample code for the Spark DataGrid with HierarchicalCollection and GroupingCollection.

Browse Flex Cookbook entries for the Spark DataGrid for new sample applications for the Spark DataGrid control.

Creating a Spark DataGrid control

You use the `<s:DataGrid>` tag to define a Spark DataGrid control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The DataGrid control uses a list-based data provider to supply the data to display. The data provider consists of a list of objects called *items*. Each item corresponds to one row of the DataGrid control. Each grid column typically corresponds to one property of each row item. For more information, see “Data providers and collections” on page 898.

You can specify data in several different ways. In the simplest case, use the `<s:ArrayList>` and `<fx:Object>` tags to define the entries. Each Object defines a row of the DataGrid control. The properties of each Object define the column entries for the row, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:DataGrid>
        <s:dataProvider>
            <s:ArrayList>
                <fx:Object>
                    <fx:Artist>Pavement</fx:Artist>
                    <fx:Price>11.99</fx:Price>
                    <fx:Album>Slanted and Enchanted</fx:Album>
                </fx:Object>
                <fx:Object>
                    <fx:Price>11.99</fx:Price>
                    <fx:Artist>Pavement</fx:Artist>
                    <fx:Album>Brighten the Corners</fx:Album>
                </fx:Object>
            </s:ArrayList>
        </s:dataProvider>
    </s:DataGrid>
</s:Application>
```

***Note:** You do not have to specify the `dataProvider` tag in MXML because `dataProvider` is the default property of the DataGrid control.*

By default, each column displays one property from each data provider item. Each column heading displays the name of the associated property in the data provider item.

The index of rows in the DataGrid control is zero-based, meaning values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items in the data provider. Column indexes are also zero-based. Therefore, if you select the first cell in the second row, the row index is 1 and the column index is 0.

Passing data to a DataGrid control

Specify the data provider for the DataGrid control by using the `dataProvider` property. The data type of the `dataProvider` property is IList. That lets you use an ArrayList, ArrayCollection, or any other class that implements IList as the data provider.

ArrayList is a lightweight implementation of IList that is useful if you do not support row sorting in the DataGrid control. To support row sorting, use a class that implements the ICollectionView interface, such as ListCollectionView. Typically, you use a subclass of ListCollectionView, such as ArrayCollection or XMLListCollection, as the data provider to support row sorting.

Each row in the data grid corresponds to one item in the data provider. Data provider items can define their properties in differing orders. By default, the order of the columns corresponds to the order of the properties as defined in the first item in the data provider.

If a data provider item omits a property or a value for a property, the DataGrid control displays an empty cell in the corresponding column for that row.

Flex lets you populate a DataGrid control from an ActionScript variable definition or from a Flex data model. The following example populates a DataGrid control by using a variable:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGPassedData.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">

    <fx:Script>
        <![CDATA[
            import mx.collections.*;
            private var dgArray:Array = [
                {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}];

            [Bindable]
            public var initDG:ArrayCollection;

            // Initialize initDG variable from the Array.
            public function initData():void {
                initDG = new ArrayCollection(dgArray);
            }
        ]]>
    </fx:Script>

    <s:DataGrid id="myGrid"
        width="350" height="200"
        dataProvider="{initDG}"/>
</s:Application>
```

In this example, you bind the variable `initDG` to the `<s:dataProvider>` property.

The following example populates the DataGrid control from XML data. In this example, you define the XML data by using the XMLList class. You then create an XMLListCollection object from the XMLList:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGXMLData.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="500">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:XMLList id="employees">
            <employee>
                <name>Joanne Wall</name>
                <phone>555-219-2012</phone>
                <email>jwall@fictitious.com</email>
                <active>true</active>
            </employee>
            <employee>
                <name>Mary Jones</name>
                <phone>555-219-2000</phone>
                <email>mjones@fictitious.com</email>
                <active>true</active>
            </employee>
            <employee>
                <name>Maurice Smith</name>
                <phone>555-219-2012</phone>
                <email>maurice@fictitious.com</email>
                <active>false</active>
            </employee>
            <employee>
                <name>Dave Davis</name>
                <phone>555-219-2000</phone>
                <email>ddavis@fictitious.com</email>
                <active>true</active>
            </employee>
            <employee>
                <name>Tom Maple</name>
                <phone>555-219-2000</phone>
                <email>tmaple@fictitious.com</email>
                <active>true</active>
            </employee>
        </fx:XMLList>
        <s:XMLListCollection id="employees2" source="{employees}"/>
    </fx:Declarations>
    <s:DataGrid id="dg" width="500" dataProvider="{employees2}">
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="name" headerText="Name"/>
                <s:GridColumn dataField="phone" headerText="Phone"/>
                <s:GridColumn dataField="email" headerText="Email"/>
            </s:ArrayList>
        </s:columns>
    </s:DataGrid>

    <s:Form>
        <s:FormItem label="Name">
            <s:Label text="{dg.selectedItem.name}"/>
        </s:FormItem>
        <s:FormItem label="Email">
            <s:Label text="{dg.selectedItem.email}"/>
        </s:FormItem>
        <s:FormItem label="Phone">
            <s:Label text="{dg.selectedItem.phone}"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

You can use Flex data access components: HTTPService, WebService, and RemoteObject, to supply data to the DataGrid. To use a remote data source to provide data, you represent the result of the remote service with the appropriate object, as follows:

- A RemoteObject component automatically returns an ArrayCollection for any data that is represented on the server as a java.util.List object, and you can use the returned object directly.
- For HTTPService and WebService results, cast the result data to a collection class if the data changes or if you use the same result in multiple places (the latter case is more efficient). As a rule, use an ArrayCollection for serialized (list-based) objects and an XMLListCollection for XML data.

The following code snippet shows this use, casting a list returned by a web service to an ArrayCollection:

```
<s:WebService id="employeeWS" wsdl="http://server.com/service.wsdl"
    showBusyCursor="true"
    fault="alert(event.fault.faultstring)">
    <s:operation name="getList">
        <mx:request>
            <deptId>{dept.selectedItem.data}</deptId>
        </mx:request>
    </s:operation>
    ...
</s:WebService>

<fx:ArrayCollection id="ac"
    source="mx.utils.ArrayUtil.toArray(employeeWS.getList.lastResult)"/>
<s:DataGrid dataProvider="{ac}" width="100%">
```

For more information on using data access component, see Accessing server-side data.

Configuring columns

A GridColumn object represents each column in a DataGrid control. By default, the DataGrid control creates a column for every property in the first item of the data provider.

Use the `DataGrid.columns` property to explicitly define the columns of the DataGrid. By explicitly defining the columns, you can set the column order, set column visibility, and set additional column properties.

The data type of the `columns` property is IList. That lets you use an ArrayList, ArrayCollection, or any other class that implements IList as the data provider. Most of the applications in this document use ArrayList to set the `columns` property.

To support column sorting, use a class that implements the ICollectionView interface, such as ListCollectionView. Typically, you use a subclass of ListCollectionView, such as ArrayCollection, as the data provider.

***Note:** You can only sort the columns of the DataGrid programmatically by sorting the IList passed to the `columns` property. That is, you cannot use the mouse to drag a column to rearrange the columns.*

Use the `GridColumn.dataField` property to specify the property of the data provider items displayed in a column, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGSpecifyColumns.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:DataGrid>
        <s:dataProvider>
            <s:ArrayCollection>
                <fx:Object Artist="Pavement" Price="11.99"
                           Album="Slanted and Enchanted"/>
                <fx:Object Artist="Pavement"
                           Album="Brighten the Corners" Price="11.99"/>
            </s:ArrayCollection>
        </s:dataProvider>
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="Album"/>
                <s:GridColumn dataField="Price"/>
            </s:ArrayList>
        </s:columns>
    </s:DataGrid>
</s:Application>
```

When you use the `<s:columns>` tag, the DataGrid only displays the columns corresponding to the specified GridColumn objects. In this example, you only define columns for the Album and Price properties of the data provider. Therefore, the DataGrid does not display a column for the Price property.

You can reorder the columns by changing the order of the GridColumn objects, as the following example shows:

```
<s:columns>
    <s:ArrayList>
        <s:GridColumn dataField="Price"/>
        <s:GridColumn dataField="Album"/>
    </s:ArrayList>
</s:columns>
```

In this example, you specify that the Price column is the first column in the DataGrid control, and that the Album column is the second.

You can also use the `<s:GridColumn>` tag to set other options. The following example uses the `headerText` property to set the name of the column to a value different from the default name of Album, and uses the `width` property to set an explicit column width:

```
<s:columns>
    <s:ArrayList>
        <s:GridColumn dataField="Album" width="200"/>
        <s:GridColumn dataField="Price" headerText="List Price"/>
    </s:ArrayList>
</s:columns>
```

Rather than explicitly set the width of a column, you can define a typical item for the DataGrid control. The DataGrid uses the typical item, and any item renderer associated with a column, to compute the initial width of each column that does not specify an explicit width. Use the `typicalItem` property to specify the data item, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGTypicalItem.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:DataGrid requestedRowCount="5">
        <s:typicalItem>
            <s:DataItem key="9999999" name="Typical name length"
                price="1234.56" call="false"/>
        </s:typicalItem>

        <s:ArrayCollection id="items">
            <s:DataItem key="1000" name="Abrasive" price="100.11" call="false"/>
            <s:DataItem key="1001" name="Brush" price="110.01" call="true"/>
            <s:DataItem key="1002" name="Clamp" price="120.02" call="false"/>
            <s:DataItem key="1003" name="Drill" price="130.03" call="true"/>
            <s:DataItem key="1004" name="Epoxy" price="140.04" call="false"/>
            <s:DataItem key="1005" name="File" price="150.05" call="true"/>
        </s:ArrayCollection>
    </s:DataGrid>
</s:Application>
```

Sorting the columns of the Spark DataGrid control

By default, the DataGrid control displays rows in the order specified in the data items passed to its `dataProvider` property. The control lets you sort data based on the cell value of a single column. To sort ascending order of a column, select the column header. Select it again to sort in descending order.

To disable sorting for the control, set the `DataGrid.sortableColumns` property to `false`. To disable sorting for an individual column, set the `GridColumn.sortable` property to `false`.

Create a custom sort by implementing a sort compare function. You can define a different sort compare function for each column of the control. For more information, see “Create a custom sort for the Spark DataGrid control” on page 571.

Hiding and displaying columns

If you display a column at some times, but not at others, you can specify the GridColumn class `visible` property to hide or show the column. The following example lets you hide or show the album price by clicking a button:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGVisibleColumn.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:DataGrid id="myDG" width="350">
        <s:ArrayCollection>
            <fx:Object Artist="Pavement" Price="11.99"
                Album="Slanted and Enchanted"/>
            <fx:Object Artist="Pavement"
                Album="Brighten the Corners" Price="11.99"/>
        </s:ArrayCollection>
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="Artist"/>
                <s:GridColumn dataField="Album"/>
                <s:GridColumn id="price" dataField="Price" visible="false"/>
            </s:ArrayList>
        </s:columns>
    </s:DataGrid>

    <!-- The column id property specifies the column to show.-->
    <s:Button label="Toggle Price Column"
        click="price.visible = !price.visible;" />
</s:Application>
```

The Spark Grid control

The Spark DataGrid control is a skinnable component that uses a Grid control as a skin part. The Grid control displays a list of data items in a two-dimensional grid of cells. Each object in the data provider of the Grid control defines a single row. Individual properties of each object define each cell of the row.

A GridColumn object represents each column of the Grid control. The GridColumn object is responsible for displaying a cell value for each row of the grid. To display a cell, the GridColumn object specifies an item renderer. To edit a cell, the GridColumn object specifies an item editor.

The Grid control only creates as many item renderers as are required to display the currently visible cells. For example, you define a Grid control with hundreds or thousands of rows. Flex only creates enough item renderers to display the currently visible cells, but not for the cells that are currently off the screen.

When a cell is moved off the visible area of the screen, its item renderer is recycled. That means the item renderer can be reused when a new cell moves onto the visible area of the screen. Recycling item renderers greatly reduces the overhead required to use the Grid control.

The following list describes some other differences between the DataGrid and Grid controls:

- Unlike the DataGrid control, the Grid control is not skinnable.
- By default, the Grid control does not display scroll bars. To add scroll bars, wrap the Grid control in the Scroller component.
- The Grid control does not provide default mouse or keyboard event handling. You must add support for those events yourself.

Handling events in a Spark DataGrid control

The DataGrid control defines event types that let you respond to user interaction. Many of these events are similar to events used by other Spark controls. For example, the DataGrid control dispatches a `gridClick` event when a user clicks any cell in the DataGrid control, similar to the `click` event used by other controls. Similarly, use the `gridMouseDown` and `gridMouseUp` events with the DataGrid control instead of the `mouseDown` and `mouseUp` events.

The DataGrid uses its own event types because it lets the DataGrid pass additional information to event handlers in the event object. For example, the DataGrid passes a spark.events.GridEvent object to the event handler of a `gridClick` event. The GridEvent class includes the cell location, and other information necessary to handle the event.

In your event handler, you often access properties of the DataGrid control. For example, you can access the DataGrid control by using the `event.currentTarget` property, where `event` is the event object passed to the event handler.

However, most of the properties of the DataGrid control are just references to properties of the underlying Grid control. Therefore, it is more efficient to access the Grid control directly in your event handler. You can access the Grid control by using the `grid` skin part of the DataGrid control, as shown below:

```
event.currentTarget.grid
```

The `grid` property in many event objects, such as event objects of type spark.events.GridEvent, also contains a reference to the underlying Grid control. Therefore, in an event handler where the event object contains a reference to the Grid control, you can reference it as follows:

```
event.grid
```

In the following example, you handle the `gridClick` event. In the event handler, you then reference the properties of the `event` object, in this case the spark.events.GridEvent class, and properties of the underlying Grid control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    minWidth="450" minHeight="450">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import spark.events.GridEvent;

            private function gridClickEvent(event:GridEvent):void {
                // Access the colunm index, row index, and event type
                // by using the GridEvent object.
                clickColumn.text = String(event.columnIndex);
                clickRow.text = String(event.rowIndex);
                eventType.text = event.type;
                // Access the selection mode of the Grid control.
                selectionType.text = String(event.grid.selectionMode);
            }
        ]]>
    </fx:Script>

    <s:DataGrid id="myGrid" width="350" height="150"
        selectionMode="multipleCells"
        gridClick="gridClickEvent(event);">
        <s:ArrayCollection>
            <fx:Object Artist="Pavement" Price="11.99"
                Album="Slanted and Enchanted" />
            <fx:Object Artist="Pavement" Album="Brighten the Corners"
                Price="11.99" />
        </s:ArrayCollection>
    </s:DataGrid>

    <s:Form>
        <s:FormItem label="Column Index:">
            <s:Label id="clickColumn"/>
        </s:FormItem>
        <s:FormItem label="Row Index:">
            <s:Label id="clickRow"/>
        </s:FormItem>
        <s:FormItem label="Selection type:">
            <s:Label id="selectionType"/>
        </s:FormItem>
        <s:FormItem label="Type:">
            <s:Label id="eventType"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

In this example, you use the event handler to display the column index, row index, event type, and selection mode.

Selecting cells and rows in the Spark DataGrid control

The DataGrid control lets you configure how the user can select cells and rows. For example, you can configure the control so that selection is disabled, or so that the user can select multiple cells of the grid, or multiple rows.

The following properties of the DataGrid control determine selection:

- `selectionMode` Specifies the selection mode of the control. Possible values are: `none` (selection disabled), `singleCell`, `singleRow` (default), `multipleCells`, and `multipleRows`.
- `requireSelection` Specifies that a cell or row must always be selected. The default value is `false`. If `true`, and the `selectionMode` property is not `none`, then the first cell or row is selected by default until the user makes a selection, or until you set the selection programmatically.

Use the keyboard or mouse to interact with the DataGrid control and to change selection. For example, when using the keyboard, use the Arrow keys to change the selected item. If you configure the DataGrid control for multiple selection, select multiple items by using the Shift and Arrow keys. For more information on keyboard navigation, see "Spark DataGrid control keyboard shortcuts" on page 573.

Handling selection events

Interactive changes to selection performed with a mouse or keyboard are processed in three steps:

1. The DataGrid control dispatches the `selectionChanging` event. In the event handler for the `selectionChanging` event, you can call the `preventDefault()` method to cancel the proposed selection change.
2. If the `selectionChanging` event is not canceled, the control dispatches a `selectionChange` event to indicate that the selection change has been committed.
3. If the selection change is committed, or if there was a change to the caret with no change to the selected item, dispatch a `caretChange` event.

You can change the selection programmatically by using methods of the DataGrid control. The control dispatches a `valueCommit` event on the change. Programmatic changes of selection are unconditionally committed, meaning that you cannot cancel them by calling the `preventDefault()` method.

More Help topics

"Working with the caret in the Spark DataGrid control" on page 558

Handle single cell and single row selection in the spark DataGrid control

Several properties of the DataGrid and Grid control handle cell and row selection in the control. These properties include the following:

- `selectedCell` If the `selectionMode` property is `singleCell`, this property contains the position of the cell in the grid as represented by the CellPosition class. The `CellPosition.columnIndex` and `CellPosition.rowIndex` properties contain the cell location.
- `selectedIndex` If the `selectionMode` property is `singleRow`, this property contains the index of the currently selected row.
- `selectedItem` The object from the data provider corresponding to the currently selected data item. This object represents the entire row of the grid.

***Note:** While the `selectedCell`, `selectedIndex`, and `selectedItem` properties are all writable, Adobe recommends that you use methods of the DataGrid to change selection programmatically.*

In the following example, when you select a row in the control, the event handler for the `selectionChange` event displays the row index and the lastName field of the data item in TextArea controls:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGSelection.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="450" width="450">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import spark.components.Grid;
            import spark.events.GridSelectionEvent;

            protected function selectionChangeHandler(event:GridSelectionEvent):void {
                const eventGrid:Grid = event.currentTarget.grid;
                var currentIndx:int = eventGrid.selectedIndex;
                var currentDataItem:Object = eventGrid.selectedItem;
                selIndex.text = String(currentIndx);
                selLName.text = String(currentDataItem.lastName);
            }
        ]]>
    </fx:Script>

    <s:DataGrid id="myDG" width="100%"
        selectionChange="selectionChangeHandler(event)">
        <s:ArrayCollection>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </s:ArrayCollection>
    </s:DataGrid>

    <s:Label text="Selected index:"/>
    <s:TextArea id="selIndex" height="50"/>

    <s:Label text="Selected Last Name:"/>
    <s:TextArea id="selLName" height="50"/>

</s:Application>
```

Handle multiple cell and multiple row selection in the Spark DataGrid control

Use the `selectionMode` property of the DataGrid control to configure it for multiple selection. The default value of the `selectionMode` property is `singleRow`, which means that you can select only a single row at a time. Set the `selectionMode` property to `multipleRows` or to `multipleCells` to select multiple rows or cells.

Several properties of the DataGrid and Grid control multiple selection. These properties include the following:

- `selectedCells` If the `selectionMode` property is `multipleCells`, a Vector of CellPosition objects containing the selected cell locations.
- `selectedIndices` If the `selectionMode` property is `multipleRows`, a Vector of integers containing the indexes of the currently selected rows.
- `selectedItems` If the `selectionMode` property is `multipleRows`, a Vector of data provider items corresponding to the currently selected rows.

Note: *While the `selectedCells`, `selectedIndices`, and `selectedItems` properties are all writable, Adobe recommends that you use methods of the DataGrid to change selection programmatically.*

For multiple selection, the set of selected cells or rows extends from the selection *anchor* to the last selected row or cell. The `Grid.anchorRowIndex` and `Grid.anchorColumnIndex` properties represent the location of the selection anchor.

The `selectedIndex` and `selectedItem` properties are valid in multiple selection mode. These properties contain information about the first cell or row selected in the list of currently selected items.

Select contiguous items in the control

1 Click the first item, either a row or cell, to select it.

2 Hold down the Shift key as you select an additional item.

- If the `selectionMode` property is set to `multipleRows`, click any cell in another row to select multiple, contiguous rows.
- If the `selectionMode` property is set to `multipleCells`, click any cell to select multiple, contiguous cells.

Select discontiguous items in the control

1 Click the first item, either a row or cell, to select it.

2 Hold down the Control key (Windows) or the Command key (OSX) as you select an additional item.

- If the `selectionMode` property is set to `multipleRows`, click any cell in another row to select that single row.
- If the `selectionMode` property is set to `multipleCells`, click any cell to select that single cell.

***Note:** You can also use the keyboard to select multiple contiguous or discontiguous items. For information on keyboard navigation, see "Spark DataGrid control keyboard shortcuts" on page 573.*

Example: Spark DataGrid using multiple selection

The following example sets the `selectionMode` property to `multipleCells`. This application uses an event handler for the `keyUp` event to recognize the Control+C key combination and, if detected, copies the selected cells from the DataGrid control to your system's clipboard.

After you copy the cells, you can paste the cells in another location in the application, or paste them in another application. In this example, you paste them to the TextArea control located at the bottom of the application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGMultiSelect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import flash.events.KeyboardEvent;
            import flash.system.System;
            import spark.components.gridClasses.CellPosition;
            import flash.ui.Keyboard;

            // Event handler to recognize when Ctrl-C is pressed,
            // and copy the selected cells to the system clipboard.
            private function myKeyUpHandler(event:KeyboardEvent):void
            {
                if (event.ctrlKey && event.keyCode == Keyboard.C)
                {
                    // Separator used between Strings sent to clipboard
                    // to separate selected cells.
                    const separator:String = ",";
                    // The String sent to the clipboard
                    var dataString:String = "";

                    // Loop over the selectedCells property.
                    // Data in selectedCells is ordered so that
                    // the last selected cell is at the head of the list.
               const selectedCells:Vector.<CellPosition> = event.currentTarget.selectedCells;
                    const n:int = selectedCells.length;
                    for (var i:int = 0; i < n; i++)
                    {
                        // Get the cell position.
                        var cell:CellPosition = selectedCells[i];
                        // Get the row for the selected cell.
                        var data:Object =
                            event.currentTarget.grid.dataProvider[cell.rowIndex];

                        // Get the name of the field for the selected cell.
                        var dataField:String =
                            event.currentTarget.grid.columns[cell.columnIndex].dataField;

                        // Get the cell data using the field name.
                        dataString = data[dataField] + separator + dataString;
                    }

                    // Remove trailing separator.
                    dataString =
                        dataString.substr(0, dataString.length - separator.length);

                    // Write dataString to the clipboard.
                    System.setClipboard(dataString);
                }
            }
        ]]>
    </fx:Script>

    <s:DataGrid
        selectionMode="multipleCells"
        keyUp="myKeyUpHandler(event);">
        <s:ArrayCollection>
            <fx:Object>
                <fx:Artist>Pavement</fx:Artist>
                <fx:Price>11.99</fx:Price>
                <fx:Album>Slanted and Enchanted</fx:Album>
            </fx:Object>
            <fx:Object>
                <fx:Artist>Pavement</fx:Artist>
                <fx:Album>Brighten the Corners</fx:Album>
                <fx:Price>11.99</fx:Price>
            </fx:Object>
        </s:ArrayCollection>
        <s:columns>
            <s:ArrayCollection>
                <s:GridColumn dataField="Artist"/>
                <s:GridColumn dataField="Album"/>
                <s:GridColumn dataField="Price"/>
            </s:ArrayCollection>
        </s:columns>
    </s:DataGrid>

    <s:TextArea id="myTA" text="Paste selected cells here ..."/>
</s:Application>
```

Working with the caret in the Spark DataGrid control

Depending on the selection mode of the DataGrid control, the *caret* is the row or cell that currently responds to keyboard input. The caret can be the currently selected cell or row, or a different cell or row. The `Grid.caretRowIndex` and `Grid.caretColumnIndex` properties represent the caret location.

Change the caret with a combination of the Control and Arrow keys. You can select the caret by pressing the Space key. If multiple selection is enabled, press the Space key to add the caret to the list of selected cells or rows. Toggle selection of the caret by using the Control-Space keys.

The following example uses the `caretChange` event and the `selectionChange` event to display the location of the caret and of the selected row:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGCaret.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="450" width="450">
    <s:layout>
        <s:VerticalLayout paddingTop="5" paddingLeft="5"/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import spark.events.GridCaretEvent;
            import spark.events.GridSelectionEvent;
            import spark.components.Grid;
            protected function selectionChangeHandler(event:GridSelectionEvent):void {
                const eventGrid:Grid = event.currentTarget.grid;
                selLabel.text = "The selected row is: " +
                    String(eventGrid.selectedIndex);
                caretLabel.text = "The caret row is: " +
                    String(eventGrid.caretRowIndex);
            }
            protected function caretChangeHandler(event:GridCaretEvent):void {
                const eventGrid:Grid = event.currentTarget.grid;
                selLabel.text = "The selected row is: " +
                    String(eventGrid.selectedIndex);
                caretLabel.text = "The caret row is: " +
                    String(eventGrid.caretRowIndex);
            }
        ]]>
    </fx:Script>

    <s:Label id="selLabel"/>
    <s:Label id="caretLabel"/>

    <s:DataGrid id="myDG" width="100%"
        selectionChange="selectionChangeHandler(event);"
        caretChange="caretChangeHandler(event);">
        <s:ArrayCollection>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </s:ArrayCollection>
    </s:DataGrid>
</s:Application>
```

To see the difference between the `selectedIndex` and the `caretIndex` properties:

1 Select a row in the DataGrid control by clicking it. Notice that the selected row is also the caret, and that the `selectedIndex` and the `caretIndex` properties have the same value.

2 Use the Up Arrow and Down Arrow keys to move the selected row. Notice that the `selectedIndex` and the `caretIndex` properties have the same value.

3 With a row selected:

- (On Windows) Hold down the Control key, and then use the Up Arrow and Down Arrow keys to move the caret.
- (On OSX) Hold down the Command key, and then use the Up Arrow and Down Arrow keys to move the caret.

Notice that the control highlights the caret row, but the index of the selected row does not change. Press the Space key to select the caret row.

Handle data binding warnings for the Spark DataGrid control

The Spark DataGrid can work with static data, meaning data that does not change at runtime. For example, you can define static data as an ArrayCollection of Objects. The following example defines static data in the DataGrid control:

```
<s:DataGrid>
    <s:dataProvider>
        <s:ArrayCollection>
            <fx:Object Artist="Pavement" Album="Slanted and Enchanted"
                Price="11.99" Cover="../assets/slanted.jpg"/>
            <fx:Object Artist="Pavement" Album="Brighten the Corners"
                Price="11.99" Cover="../assets/brighten.jpg"/>
        </s:ArrayCollection>
    </s:dataProvider>
...
</s:DataGrid>
```

Dynamic data is data that can change at runtime. When dynamic data changes, you want to ensure that the DataGrid recognizes those changes so that it updates its display accordingly.

The item renderer and item editor mechanism used by the Spark DataGrid control relies on data binding to items in the data provider. Data binding is event driven. That means an item in the data provider must be able to dispatch events to indicate that the item has changed. By using the data binding mechanism, the DataGrid control can update its display at runtime when an item changes.

In the previous example that used static data, you used the Object class to define the items of the data provider. The Object class does not dispatch events when it changes and, therefore, does not support data binding. The compiler recognizes this situation and issues a warning when you compile the application. You can ignore that warning for applications that use static data.

One way to add event support to items in the data provider is to make sure the class that defines the items is a subclass of the flash.events.EventDispatcher class. Or, ensure that the class implements the event mechanism.

Flex provides the spark.utils.DataItem class that is a subclass of Object and supports events. Therefore, you can rewrite the previous example to use DataItem and support data binding, as shown below:

```
<s:DataGrid>
    <s:dataProvider>
        <s:ArrayCollection>
            <fx:DataItem Artist="Pavement" Album="Slanted and Enchanted"
                Price="11.99" Cover="../assets/slanted.jpg"/>
            <fx:DataItem Artist="Pavement" Album="Brighten the Corners"
                Price="11.99" Cover="../assets/brighten.jpg"/>
        </s:ArrayCollection>
    </s:dataProvider>
...
</s:DataGrid>
```

Because the DataItem class can dispatch events when the data changes, you can modify the data at runtime and have those changes reflected in the DataGrid.

Alternatively, you can represent your data by creating a class that includes the `[Bindable]` metadata tag in the class definition. When you include the `[Bindable]` metadata tag, Flex automatically adds support for event dispatching to the class.

For example, you create the following subclass of Object that supports data binding by using the `[Bindable]` metadata tag:

```
package myComponents
{
    [Bindable]
    public class MyBindableObj extends Object
    {
        public function MyBindableObj() {
            super();
        }

        public var Artist:String;

        public var Album:String;

        public var Price:Number;

        public var Cover:String;
    }
}
```

By inserting the `[Bindable]` metadata tag before the class definition, you specify that all public properties support data binding. You can then use your custom class to define the data provider, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/SparkDGBindingClass.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:myComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:DataGrid>
        <s:dataProvider>
            <s:ArrayCollection>
                <myComp:MyBindableObj Artist="Pavement"
                    Album="Slanted and Enchanted"
                    Price="11.99"
                    Cover="../assets/slanted.jpg"/>
                <myComp:MyBindableObj Artist="Pavement"
                    Album="Brighten the Corners"
                    Price="11.99"
                    Cover="../assets/brighten.jpg"/>
            </s:ArrayCollection>
        </s:dataProvider>
    </s:DataGrid>
</s:Application>
```

More Help topics

"Data binding" on page 299

"Bindable metadata tag" on page 2381

Creating item renderers and item editors for the Spark DataGrid control

You can create custom item renderers and item editors to control the display of the cells of the Spark DataGrid and Grid controls. You can also create custom item renderers for the header cells of each column of the grid.

You set a custom renderer or editor for each column of the grid. Use the `GridColumn.itemRenderer`, `GridColumn.itemEditor`, and `GridColumn.headerRenderer` properties to specify your custom renderer or editor.

You typically create custom item renderers and item editors in MXML. However, for the highest performance, create them in ActionScript.

Adobe Engineer Joan Lafferty creates a custom item renderer for the Spark DataGrid control in Using a DateField itemRenderer for a Spark DataGrid

The IGridItemRenderer and IGridItemEditor interfaces

Item and header renderers must implement the IGridItemRenderer interface. Item editors must implement the IGridItemEditor interface. The following table describes these interfaces:

Interface	Implemented by	Notes
IGridItemRenderer	Custom item and header renderers	All custom item and header renderers must implement the IGridItemRenderer interface. The base interface of IGridItemRenderer is IDataRenderer, the interface implemented by item renderers for the Spark list-based controls.
IGridItemEditor	Custom item editors	All custom item editors must implement the IGridItemEditor interface. The base interface of IGridItemEditor is IDataRenderer.

Using the predefined item renderers

When using the DataGrid control, you want to ensure the highest performance possible. One of the main factors affecting performance is the time required to render of each visible cell of the grid. Therefore, you want your item renderers to perform at the highest level.

Flex defines several item renderers that you can use to achieve high performance with the DataGrid control. By default, the DataGrid control uses the DefaultGridItemRenderer renderer. This renderer is written in ActionScript to provide good performance across all platforms.

The following table describes the item renderer classes that ship with Flex:

Item Renderer	Use
spark.skins.spark.DefaultGridItemRenderer	The default item renderer that displays the cell data in a text label using the UIFTETextField control. This class is not intended to be extended. Create a custom item renderer based on the GridItemRenderer class. Because it supports the Flash Text Engine (FTE), this item render also supports bidirectional text.
spark.skins.spark.UITextFieldGridItemRenderer	Optimized for deployment on Microsoft Windows. For Windows, it provides improved performance over the DefaultGridItemRenderer. For other operating systems, use DefaultGridItemRenderer for best performance. Because it is based on the TextField component, this item renderer does not support bidirectional text.
spark.components.gridClasses.GridItemRenderer	The base class for custom item renderers. This class implements the IGridItemRenderer interface.

Creating an item renderer for a Spark DataGrid

Item renderers for the Spark list-based controls, such as List, must implement the IDataRenderer interface. Item renderers for the Spark DataGrid must implement the IGridItemRenderer interface, which is derived from the IDataRenderer interface. Therefore, the process of creating an item renderer for the DataGrid is similar to creating one for the List control.

An item renderer is associated with a column of the DataGrid control. The item renderer then controls the appearance of each cell in the column. However, each item renderer has access to the data item for an entire row of the DataGrid control. Use the `data` property of the item renderer to access the data item.

By accessing the data item for the entire row, the item renderer can display multiple data fields, or display a single value created from multiple fields. For example, each row of the data provider for the DataGrid control in the following example contains four fields:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGComplexIR.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="450" height="450">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Data includes URL to album cover.
            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover:'assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover:'assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>

    <s:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="Artist"/>
                <s:GridColumn dataField="Album" itemRenderer="myComponents.DGComplexIR"/>
                <s:GridColumn dataField="Price"/>
            </s:ArrayList>
        </s:columns>
    </s:DataGrid>
</s:Application>
```

However, the DataGrid control only defines three columns. It then uses the DGComplexIR.mxml item renderer to display the Album and Cover fields in a single column. The item render is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\myComponents\DGComplexIR.mxml -->
<s:GridItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:VGroup height="100" paddingTop="10">
        <s:Label id="albumName"
            width="100%"
            text="{data.Album}"/>
        <s:Image id="albumImage"
            source="{data.Cover}"/>
    </s:VGroup>
</s:GridItemRenderer>
```

Notice how the item renderer uses the `data` property of the item renderer to access the data item that corresponds to the entire row of the DataGrid control.

Creating inline renderers

In the previous example, the item renderer was defined in a file separate from the main application file. You can define inline item renderers for the DataGrid control. By using an inline item renderer, your code can all be defined in a single file. To define an inline item renderer, you use the `<fx:Component>` tag. For example using inline item renderers, see "Defining an inline item renderer for a Spark container" on page 491.

The following example creates an inline item renderer by using the Spark Label control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGStyledIR.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="450" height="450">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Data includes URL to album cover.
            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover:'../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover:'../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>

    <s:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="Artist">
                    <s:itemRenderer>
                        <fx:Component>
                            <s:GridItemRenderer>
                                <s:Label id="labelDisplay" fontSize="24"/>
                            </s:GridItemRenderer>
                        </fx:Component>
                    </s:itemRenderer>
                </s:GridColumn>
                <s:GridColumn dataField="Album"/>
                <s:GridColumn dataField="Price"/>
            </s:ArrayList>
        </s:columns>
    </s:DataGrid>
</s:Application>
```

Creating a header renderer for a Spark DataGrid

The DataGridSkin class uses the GridColumnHeaderGroup component to control the display of the column headers. The GridColumnHeaderGroup component displays a row of headers cells, where the vertical edges of the header cells are aligned with the grid columns.

The GridColumnHeaderGroup component arranges header renderer instances in a row, where the left and right edge of each renderer match the corresponding column. The renderers' height is the maximum preferred height for all of the displayed header renderers.

Like the Grid control and item renderers, the GridColumnHeaderGroup class only creates as many column header renderers and separators as are visible. Renderers and separators that have been scrolled out of view are recycled.

By default, the GridColumnHeaderGroup component uses the spark.skins.spark.DefaultGridHeaderRenderer class as the header renderer. To create a custom header renderer, define the renderer in MXML or ActionScript. Header renderers must implement the IGridItemRenderer interface.

Then, use the `Grid.headerRenderer` property to specify to use the custom renderer for the column.

Creating an item editor for a Spark DataGrid

The DataGrid control includes an `editable` property that you set to `true` to let users edit grid cells. By default, the value of the `editable` property is `false`, which means that you cannot edit the cells.

For a DataGrid control, setting the `editable` property to `true` enables editing for all columns of the grid. You can disable editing for any column by setting the `GridColumn.editable` property to `false`.

Item editing is cell based. To edit a cell, first select the cell or cell row, depending on the selection mode of the grid. Then select the cell to edit. If editing is enabled, an item editor appears over the selected cell.

***Note:** While an item editor is associated with a single cell, the editor actually has access to the data provider element for the entire row of the selected cell. Therefore, the item editor can access and modify any data for the row.*

All item editors must implement the IGridItemEditor interface. The GridItemEditor class implements the IGridItemEditor interface. GridItemEditor also adds the `value` property that you can use to pass data to and from the item editor. Most custom item editors are created as subclasses of GridItemEditor.

The two most important questions when dealing with an item editor are:

- How do you pass data to the item editor?

  Use the bindable `GridItemEditor.value` property to pass data to the item editor. The data type of the `value` property is Object, so you can use it to pass a single value to the item editor, or you can use it to pass multiple items as fields of the Object.

  The GridItemEditor class implements the `value` property as a setter and a getter method. When the item editor is created, Flex calls the setter method, passing the cell data from the data provider item for the row. Typically, you override the setter method in your item editor to initialize any items in the item editor from the cell data. The `value` property only exists while the item editor is open.

  To access the data provider element for the entire row, use the `data` property of the item renderer.

- How do you pass data back to the DataGrid control from the item editor?

  Use the `GridItemEditor.value` property to pass data back to the control. Typically, in your item editor you override the getter method for the `value` property to return any results back to the DataGrid control. The DataGrid control writes the returned value to the data field of the data provider element for the row that corresponds to the edited cell.

  The item editor uses the `IGridItemEditor.save()` method to write the `value` property to the data provider of the DataGrid control. You can override the `save()` method to control how the `value` property is written to the data provider. For example, the item editor might return multiple values that you want to write to multiple fields of the data provider element for the row. Override the `save()` method in this situation to update the data provider.

Using the predefined item editors

Flex ships with two item editors that you can use in your application. The following table describes these item editor classes:

Item Editor	Use
spark.components.gridClasses.DefaultGridItemEditor	Uses a Spark TextArea control to let you edit the text value of a cell. By default, the DataGrid control uses the DefaultGridItemEditor class.
spark.components.gridClasses.ComboBoxGridItemEditor	Uses a Spark ComboBox control to display a drop-down list of cell values. Select a value to set the new value of the cell.

The following example shows a DataGrid control that uses the ComboBoxGridItemEditor class as the item editor:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGComboBoxIE.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               width="450">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, color:'red'},
                {label1:"Order #2315", quant:3, color:'red'}
            ]);
        ]]>
    </fx:Script>

    <s:DataGrid id="myDG" width="100%"
                dataProvider="{myDP}"
                variableRowHeight="true"
                editable="true" >
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="label1" headerText="Order #"/>
                <s:GridColumn dataField="quant" headerText="Qty"/>
                <s:GridColumn dataField="color" headerText="Color">
                    <s:itemEditor>
                        <fx:Component>
                            <s:ComboBoxGridItemEditor>
                                <s:dataProvider>
                                    <s:ArrayList>
                                        <fx:String>red</fx:String>
                                        <fx:String>green</fx:String>
                                        <fx:String>blue</fx:String>
                                    </s:ArrayList>
                                </s:dataProvider>
                            </s:ComboBoxGridItemEditor>
                        </fx:Component>
                    </s:itemEditor>
                </s:GridColumn>
            </s:ArrayList>
        </s:columns >
    </s:DataGrid>
</s:Application>
```

The item editing process

The following steps describe the lifecycle of an item editor:

1 An editing session begins in response to a user gesture, such as the user clicking a selected cell. You can also make an explicit call to the `DataGrid.startItemEditorSession()` method to start the editing session.

2 Before the item editor appears, the DataGrid dispatches the `gridItemEditorSessionStarting` event.

You can cancel the editing session by calling the `preventDefault()` method in the event handler for the `gridItemEditorSessionStarting` event.

3 Flex sets the `rowIndex`, `column`, and `data` properties of the item editor. The `data` property contains the data provider item for the entire row.

Flex sets the `value` property of the item editor to the value of the field in the data provider element that corresponds to the cell being edited.

4 Flex calls the `IGridItemEditor.prepare()` method.

When the `prepare()` method is called, the editor's size and location have been set and the editor's layout has been validated. You can override the `prepare()` method to make any final modifications to the editor, such as modifying its visual characteristics or attaching event handlers.

5 Flex makes the editor visible by setting its `visible` property to `true`.

6 The DataGrid dispatches the `gridItemEditorStart` event.

7 When the item editor is first displayed, it is given keyboard focus and its `setFocus()` method is called. You typically override the `setFocus()` method of the item editor to shift the focus to a specific component in the item editor.

8 The user interacts with the editor.

9 The edit session ends when the user presses the Enter key to save the data, or the Escape key to cancel the edit. The session also ends, and the data is saved, when the users clicks outside the editor, or the editor loses keyboard focus.

If the editing session is saved, Flex calls the `IGridtemEditor.saved()` method to write the `value` property back to the data provider.

You can end the editing sessions programmatically by calling the `DataGrid.endEditorSession()` method.

10 Flex calls the `IGridtemEditor.discard()` method.

Override the `discard()` method to reverse any settings you made in the `prepare()` method, such as removing event handles, or perform any other cleanup.

11 The DataGrid dispatches either the `gridItemEditorSessionSave` or `gridItemEditorSessionCancel` event.

Example custom item editor

The following example shows a DataGrid control that uses and item editor to let the user change the value of the quant field of the data provider. The second column of the control specifies to use the DGNumStepperEditor.mxml item editor:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGItemEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="450">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </fx:Script>

    <s:DataGrid id="myDG" width="100%"
        dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="label1" headerText="Order #"/>
                <s:GridColumn dataField="quant"
                    headerText="Qty"
                    itemEditor="myComponents.DGNumStepperEditor"/>
            </s:ArrayList>
        </s:columns >
    </s:DataGrid>
</s:Application>
```

The DGNumStepperEditor.mxml item editor defines a NumericStepper control to set an integer value. The item editor overrides the setter method for the `value` property to initialize the NumericStepper control with the current value of the cell.

The override of the getter method for the `value` property returns the current value of the NumericStepper control. The `save()` method updates the data provider element for the row of the grid with this value. The DGNumStepperEditor.mxml item editor is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\myComponents\DGNumStepperEditor.mxml -->
<s:GridItemEditor xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[

            // Override the setter to initialize the NumericStepper control
            // with the cell data.
            override public function set value(newValue:Object):void {
                ns.value = newValue as Number;
            }
            // Override the getter to return the current value of
            // the NumericStepper control.
            // The save() method updates the data provider element for the
            // row of the grid with this value.
            override public function get value():Object {
                return ns.value;
            }

            // Override setFocus() to shift focus to the NumericStepper.
            override public function setFocus():void {
                ns.setFocus();
            }
        ]]>
    </fx:Script>
    <s:NumericStepper id="ns" width="100%"
        fontWeight="bold"/>
</s:GridItemEditor>
```

Keyboard and mouse shortcuts with item editors for the Spark DataGrid

The following table describes how the item editor responds to user interaction:

User interaction	Response
F2	If cell selection mode is enabled, edit the cell indicated by the caret. If row selection mode is enabled, and a cell has not been edited before, edit the first visible cell in the row indicated by the caret. Otherwise edit the cell in the last edited column in the caret row.
Enter	Save the editor's value and close the editor.
Esc	Cancel the editing session by closing the editor without saving the value.
Tab	Save the editor's value, close the editor, and open the editor in the next column that is editable.
Shift+Tab	Save the editor's value, close the editor, and open the editor in the first previous column that is editable.
Ctrl+.	Cancel the editing session by closing the editor without saving the value.
Ctrl+Enter	Save the editor's value, close the editor, and move selection to the same column in the next row.
Ctrl+Shift+Enter	Save the editor's value, close the editor, and move selection to the same column in the previous row.
Single mouse click	Open the editor in response to a single click on the selected cell. The cell must already be selected. If the Shift or Ctrl key is pressed when the click occurs, then the editor does not open because the Shift and Ctrl keys are used to modify the selection.

Create a custom sort for the Spark DataGrid control

By default, users can sort the row of a DataGrid by clicking the column headers. Clicking the column header initially sorts the display in descending order of the entries in the selected column, and clicking the header again reverses the sort order. For an example, see "Sorting the columns of the Spark DataGrid control" on page 551.

The GridColumn`sortCompareFunction` property lets you specify a custom comparison function used to sort the rows by that column. This property sets the `compareFunction` property of the default Spark SortField class object used by the DataGrid.

The `sortCompareFunction` property lets you specify the function that compares two objects and determines which would be higher in the sort order, without requiring you to explicitly create a Sort object on your data provider.

The comparison function passed to the `sortCompareFunction` property must have the following signature:

```
function sortCompareFunction(obj1:Object, obj2:Object, gc:GridColumn):int {
    // Sort logic
}
```

The `obj1` and `obj2` arguments specify the objects to compare, and `gc` specifies the column of the DataGrid control to sort.

Flex ships with two collator classes to help you write your comparison function. The spark.globalization.MatchingCollator and spark.globalization.SortingCollator provide locale-sensitive string comparison functions.

By default, the sorting used by the DataGrid control is case sensitive. In the following example, you use the SortingCollator class to perform a case-insensitive comparison of two cells to sort the Name column of the DataGrid:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dpcontrols\sparkdpcontrols\SparkDGXMLSort.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="500" height="600">
    <s:layout>
        <s:VerticalLayout paddingTop="10"/>
    </s:layout>
    <fx:Declarations>
        <fx:XMLList id="employees">
            <employee>
                <name>Joanne Wall</name>
                <phone>555-219-2012</phone>
                <email>jwall@fictitious.com</email>
                <active>true</active>
            </employee>
            <employee>
                <name>Mary Jones</name>
                <phone>555-219-2000</phone>
                <email>mjones@fictitious.com</email>
                <active>true</active>
            </employee>
            <employee>
                <name>mary jones</name>
                <phone>555-219-2000</phone>
                <email>mjones@fictitious.com</email>
                <active>true</active>
            </employee>
            <employee>
```

```
                <name>Maurice Smith</name>
                <phone>555-219-2012</phone>
                <email>maurice@fictitious.com</email>
                <active>false</active>
            </employee>
            <employee>
                <name>Dave Davis</name>
                <phone>555-219-2000</phone>
                <email>ddavis@fictitious.com</email>
                <active>true</active>
            </employee>
            <employee>
                <name>Tom Maple</name>
                <phone>555-219-2000</phone>
                <email>tmaple@fictitious.com</email>
                <active>true</active>
            </employee>
        </fx:XMLList>
        <s:XMLListCollection id="employees2" source="{employees}"/>
        <s:XMLListCollection id="employees3" source="{employees}"/>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import mx.events.FlexEvent;
            import spark.collections.Sort;
            import spark.collections.SortField;
            import spark.globalization.SortingCollator;

            // Create an instance of the SortingCollator.
            private var collator:SortingCollator = new SortingCollator();

            // Define the sort compare function used by the first column.
          private function sortCompareFunction(obj1:Object, obj2:Object, gc:GridColumn):int {
                // Make the sort case insensitive. The default is case sensitive.
                collator.ignoreCase = true;
                return collator.compare(obj1[gc.dataField], obj2[gc.dataField]);
            }
        ]]>
    </fx:Script>
    <s:Label text="Custom case insensitive sort of the Name colum"/>
    <s:DataGrid id="dg" width="500" dataProvider="{employees2}">
```

```
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="name" headerText="Name"
                    sortCompareFunction="sortCompareFunction"/>
                <s:GridColumn dataField="phone" headerText="Phone"/>
                <s:GridColumn dataField="email" headerText="Email"/>
            </s:ArrayList>
        </s:columns>
    </s:DataGrid>
    <s:Label text="Default case sensitive sort of the Name colum"/>
    <s:DataGrid width="500" dataProvider="{employees3}">
        <s:columns>
            <s:ArrayList>
                <s:GridColumn dataField="name" headerText="Name"/>
                <s:GridColumn dataField="phone" headerText="Phone"/>
                <s:GridColumn dataField="email" headerText="Email"/>
            </s:ArrayList>
        </s:columns>
    </s:DataGrid>
</s:Application>
```

Spark DataGrid control keyboard shortcuts

A DataGrid control displays a grid of rows and columns. Depending on the available screen space, all rows and columns might not be currently visible. Paging down through the data items displayed by a 10-row control shows items 0-9, 10-19, 20-29, and so on.

The user clicks individual data items to select them. If the DataGrid is configured for multiple selection, holds down Shift key while clicking to select multiple items.

The control has the following default keyboard navigation features:

Key	Action
Up Arrow	Moves selection up one item. Hold down the Control key (Windows) or Command key (OSX) to only move the caret, but not change selection.
Down Arrow	Moves selection down one item. Hold down the Control key (Windows) or Command key (OSX) to only move the caret, but not change selection.
Left Arrow	In cell selection mode, moves selection one column to the left. Hold down the Control key (Windows) or Command key (OSX) to only move the caret, but not change the selected cell.
Right Arrow	In cell selection mode, moves selection one column to the right. Hold down the Control key (Windows) or Command key (OSX) to only move the caret, but not change the selected cell.
Page Up	Moves selection up one page.
Page Down	Moves selection down one page.
Home	Moves selection to the top of the grid.
End	Moves selection to the bottom of the grid.
Control key (Windows) and Command key (OSX) and a navigation key	Allows for multiple, noncontiguous selection and deselection in multiple selection mode. Works with Arrow keys, Page Up key, Page Down key, mouse click, and mouse drag.
Shift key and a navigation key	Allows for contiguous selection in multiple selection mode. Works with keypresses, click selection, and drag selection. Works with Arrow keys, Page Up key, Page Down key, mouse click, and mouse drag.

Key	Action
Control-A	Select all grid items
Space	Adds the caret to the selection list
Control-Space	Toggle the selection of the caret.
Shift-Space	Extends selection from the anchor to the caret

MX layout containers

In Adobe® Flex®, MX layout containers provide a hierarchical structure to arrange and configure the components, such as Button and ComboBox controls, of a Flex application.

For detailed information on how Flex lays out containers and their children, see “Laying out components” on page 359.

Flex also provides a set of Spark containers. These containers are designed to simplify skinning and optimize performance. Adobe recommends that you use the Spark containers when possible. For more information, see “Spark containers” on page 417.

About layout containers

A layout container defines a rectangular region of the Adobe® Flash® Player drawing surface and controls the sizing and positioning of the child controls and child containers defined within it. For example, a Form layout container sizes and positions its children in a layout similar to an HTML form.

To use a layout container, you create the container, and then add the components that define your application.

MX Canvas layout container

A Canvas layout container defines a rectangular region in which you place child containers and controls. Unlike all other components, you cannot let Flex lay child controls out automatically. You must use *absolute* or *constraint-based* layout to position child components. With absolute layout you specify the *x* and *y* positions of the children; with constraint-based layout you specify side, baseline, or center anchors. For detailed information on using these layout techniques, see “Laying out components” on page 359.

***Note:** Adobe recommends that, when possible, you use the Spark containers with BasicLayout instead of the MX Canvas container. For more information, see “Spark containers” on page 417.*

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

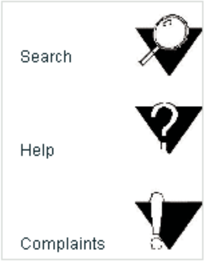
Creating and using a Canvas control

You define a canvas control in MXML by using the `<mx:Canvas>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

Creating a Canvas Control by using absolute positioning

You can use the x and y properties of each child to specify the child's location in the Canvas container. These properties specify the *x* and *y* coordinates of a child relative to the upper-left corner of the Canvas container, where the upper-left corner is at coordinates (0,0). Values for the *x* and *y* coordinates can be positive or negative integers. You can use negative values to place a child outside the visible area of the container, and then use ActionScript to move the child to the visible area, possibly as a response to an event.

The following example shows a Canvas container with three LinkButton controls and three Image controls:



The following MXML code creates this Canvas container:

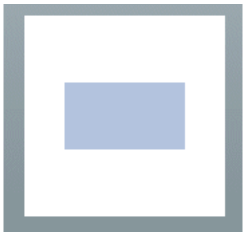
```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Canvas id="myCanvas"
      height="200" width="200"
      borderStyle="solid"
      backgroundColor="white">
      <mx:LinkButton label="Search"
        x="10" y="30"
        click="navigateToURL(new
URLRequest('http://www.adobe.com/cfusion/search/index.cfm'))"/>
      <mx:Image
        height="50" width="50"
        x="100" y="10"
        source="@Embed(source='assets/search.jpg')"
        click="navigateToURL(new
URLRequest('http://www.adobe.com/cfusion/search/index.cfm'))"/>
      <mx:LinkButton label="Help"
        x="10" y="100"
        click="navigateToURL(new URLRequest('http://www.adobe.com/go/gn_supp'))"/>
      <mx:Image
        height="50" width="50"
        x="100" y="75"
        source="@Embed(source='assets/help.jpg')"
        click="navigateToURL(new URLRequest('http://www.adobe.com/go/gn_supp'))"/>
      <mx:LinkButton label="Complaints"
        x="10" y="170"
        click="navigateToURL(
            new URLRequest('http://www.adobe.com/go/gn_contact'))"/>
      <mx:Image
        height="50" width="50"
        x="100" y="140"
        source="@Embed(source='assets/complaint.jpg')"
        click="navigateToURL(
            new URLRequest('http://www.adobe.com/go/gn_contact'))"/>
    </mx:Canvas>
</s:Application>
```

Creating a Canvas container by using constraint-based layout

You can also use *constraint-based layout* to anchor any combination of the top, left, right, bottom sides, and baseline of a child at a specific distance from the Canvas edges, or to anchor the horizontal or vertical center of the child at a specific (positive or negative) pixel distance from the Canvas center. To specify a constraint-based layout, use the `top`, `bottom`, `left`, `right`, `baseline`, `horizontalCenter`, and `verticalCenter` styles. When you anchor the top and bottom, or the left and right sides of the child container to the Canvas sides, if the Canvas control resizes, the children also resize. The following example uses constraint-based layout to position an HBox horizontally, and uses absolute values to specify the vertical width and position:

```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasConstraint.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    backgroundColor="gray"
    height="200" width="200">
    <mx:Canvas
        width="150" height="150"
        horizontalCenter="0" verticalCenter="0"
        backgroundColor="#FFFFFF">
        <mx:HBox id="hBox2"
            left="30"
            right="30"
            y="50"
            height="50"
            backgroundColor="#A9C0E7">
        </mx:HBox>
    </mx:Canvas>
</s:Application>
```

The example produces the following image:



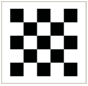
Preventing overlapping children

When you use a Canvas container, some of your components may overlap, because the Canvas container ignores its children's sizes when it positions them. Similarly, children components may overlap any borders or padding, because the Canvas container does not adjust the coordinate system to account for them.

In the following example, the size and position of each component is carefully calculated to ensure that none of the components overlap:

```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasOverlap.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="100" height="100">
    <fx:Script>
        <![CDATA[
            [Embed(source="assets/BlackBox.jpg")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </fx:Script>

    <mx:Canvas id="chboard" backgroundColor="#FFFFFF">
        <mx:Image source="{imgCls}"
            width="10" height="10" x="0" y="0"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="20" y="0"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="40" y="0"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="10" y="10"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="30" y="10"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="0" y="20"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="20" y="20"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="40" y="20"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="10" y="30"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="30" y="30"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="0" y="40"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="20" y="40"/>
        <mx:Image source="{imgCls}"
            width="10" height="10" x="40" y="40"/>
    </mx:Canvas>
</s:Application>
```

This example produces the following image:



If you set the `width` and `height` properties of one of the images to 20 pixels but don't change the positions accordingly, that image overlaps other images in the checkerboard. For example, if you replace the seventh `<mx:Image>` tag in the preceding example with the following line, the resulting image looks like the following image:

```
<mx:Image source="{imgCls}" width="20" height="20" x="20" y="20"/>
```

Repositioning children at run time

You can build logic into your application to reposition a child of a Canvas container at run time. For example, in response to a button click, the following code repositions an input text box that has the `id` value `text1` to the position x=110, y=110:

```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasRepos.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Canvas
        width="300" height="185"
        backgroundColor="#FFFFFF">
        <mx:TextInput id="text1"
            text="Move me"
            x="10" y="10"
        />
        <mx:Button id="button1"
            label="Move text1"
            x="10" y="150"
            click="text1.x=110; text1.y=110;"
        />
        <mx:Button label="Reset"
            click="text1.x=10; text1.y=10;" x="111" y="150"/>
    </mx:Canvas>
</s:Application>
```

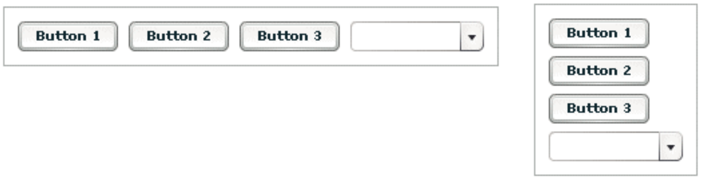
MX Box, HBox, and VBox layout containers

The Box layout container lays out its children in a single vertical column or a single horizontal row. You use the `direction` property of a Box container to determine either vertical (default) or horizontal layout. The HBox and VBox containers are Box containers with `horizontal` and `verticaldirection` property values.

Note: *Adobe recommends that, when possible, you use the Spark containers with HorizontalLayout or VerticalLayout instead of the MX box containers. For more information, see "Spark containers" on page 417.*

To lay out children in multiple rows or columns, use a Tile or Grid container. For more information, see "MX Tile layout container" on page 612 and "MX Grid layout container" on page 603.

The following example shows one Box container with a horizontal layout and one with a vertical layout:



For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a Box, HBox, or VBox container

You use the `<mx:Box>`, `<mx:VBox>`, and `<mx:HBox>` tags to define Box containers. Use the VBox (vertical box) and HBox (horizontal box) containers as shortcuts so you do not have to specify the `direction` property in the Box container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example creates a Box container with a vertical layout:

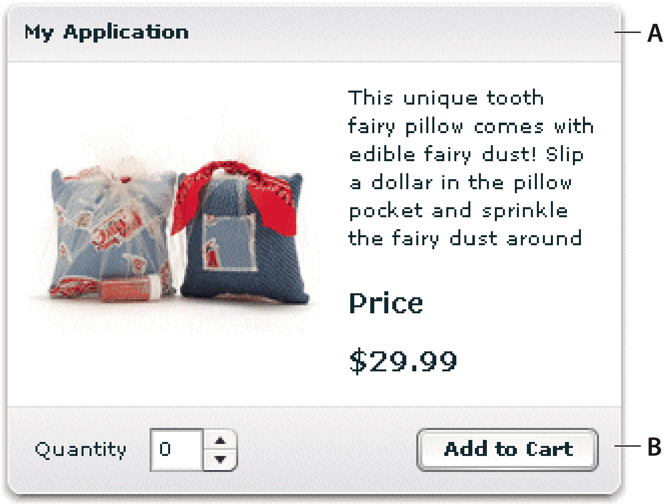
```
<?xml version="1.0"?>
<!-- containers\layouts\BoxSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Box direction="vertical"
        borderStyle="solid"
        paddingTop="10"
        paddingBottom="10"
        paddingLeft="10"
        paddingRight="10">
        <mx:Button id="fname" label="Button 1"/>
        <mx:Button id="lname" label="Button 2"/>
        <mx:Button id="addr1" label="Button 3"/>
        <mx:ComboBox id="state">
              <mx:ArrayList>
                 <fx:String>ComboBox 1</fx:String>
              </mx:ArrayList>
        </mx:ComboBox>
    </mx:Box>
</s:Application>
```

The following code example is equivalent to the previous example, except that this example defines a vertical Box container by using the `<mx:VBox>` tag:

```
<?xml version="1.0"?>
<!-- containers\layouts\VBoxSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:VBox borderStyle="solid"
        paddingTop="10"
        paddingBottom="10"
        paddingLeft="10"
        paddingRight="10">

        <mx:Button id="fname" label="Button 1"/>
        <mx:Button id="lname" label="Button 2"/>
        <mx:Button id="addr1" label="Button 3"/>
        <mx:ComboBox id="state">
              <mx:ArrayList>
                 <fx:String>ComboBox 1</fx:String>
              </mx:ArrayList>
        </mx:ComboBox>
    </mx:VBox>
</s:Application>
```

MX ControlBar layout container

You use the MX ControlBar container with a MX Panel or MX TitleWindow container to hold components that can be shared by the other children in the Panel or TitleWindow container. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart, as the following example shows:


My Application
This unique tooth fairy pillow comes with edible fairy dust! Slip a dollar in the pillow pocket and sprinkle the fairy dust around
Price
$29.99
Quantity
0
Add to Cart
A
B

A. Panel container *B.* ControlBar container

Note: *You cannot use the MX ControlBar container with the Spark Panel container. The Spark Panel container defines the controlBarContent and controlBarLayout properties that you use the add this functionality. For more information, see "The Spark Panel container" on page 434*

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating an MX ControlBar container

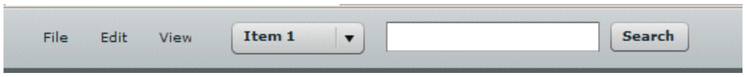
You use the `<mx:ControlBar>` tag to define a ControlBar control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML code, either in another tag or in an ActionScript block. You specify the `<mx:ControlBar>` tag as the last child tag of an `<mx:Panel>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\CBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            private function addToCart():void {
                // Handle event.
            }
        ]]>
    </fx:Script>
    <mx:Panel title="My Application"
        paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10">
        <mx:HBox width="250" height="200">
            <!-- Area for your catalog. -->
        </mx:HBox>

        <mx:ControlBar width="250">
            <mx:Label text="Quantity"/>
            <mx:NumericStepper/>
            <!-- Use Spacer to push Button control to the right. -->
            <mx:Spacer width="100%"/>
            <mx:Button label="Add to Cart"
                click="addToCart();"/>
        </mx:ControlBar>
    </mx:Panel>
</s:Application>
```

MX ApplicationControlBar layout container

You use the MX ApplicationControlBar container to hold components that provide access to application navigation elements and commands. An ApplicationControlBar container for an editor, for example, could include Button controls for setting the font weight, a ComboBox to select the font, and a MenuBar control to select the edit mode. The ApplicationControlBar is a subclass of the ControlBar class; however, it has a different look and feel.

Typically, you place an ApplicationControlBar container at the top of the application, as the following example shows:



If you dock the ApplicationControlBar container at the top of an application, it does not scroll with the application contents.

***Note:** You cannot use the MX ApplicationControlBar container with the Spark Application container. The Spark Application container defines the controlBarContent and controlBarLayout properties that you use the add this functionality. For more information, see “Adding a control bar area to the application container” on page 400*

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating an ApplicationControlBar container

You use the `<mx:ApplicationControlBar>` tag to define a ControlBar control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML code, either in another tag or in an ActionScript block.

The ApplicationControlBar container can be in either of the following modes:

Docked mode The bar is always at the top of the application's drawing area. Any application-level scroll bars don't apply to the container, so it always remains at the top of the visible area, and the bar expands to fill the width of the application. To created a docked ApplicationControlBar container, set its `dock` property to `true`.

Normal mode The bar can be placed anywhere in the application, is sized and positioned just like any other component, and scrolls with the application. The ApplicationControlBar floats if its `dock` property is `false`. The default value is `false`.

***Note:** In contrast to the ControlBar container, it is possible to set the `backgroundColor` style for an instance of the ApplicationControlBar. The ApplicationControlBar container has two styles, `fillColors` and `fillAlpha`, that are not supported by the ControlBar container.*

The following example shows an application with a simple docked ApplicationControlBar that includes a MenuBar. The Application also includes an HBox control that exceeds the application size; when you scroll the application to view the bottom of the HBox control, the ApplicationControlBar control does not scroll.

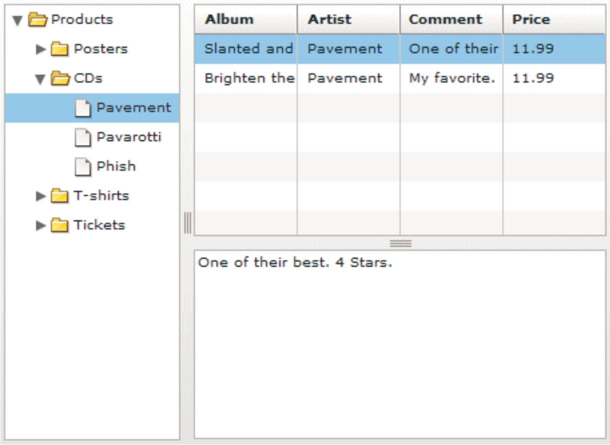
```
<?xml version="1.0"?>
<!-- containers\layouts\AppCBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="550">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:XMLList id="menuXML">
            <fx:menuitem label="File">
                <fx:menuitem label="New" data="New"/>
                <fx:menuitem label="Open" data="Open"/>
                <fx:menuitem label="Save" data="Save"/>
                <fx:menuitem label="Exit" data="Exit"/>
            </fx:menuitem>
            <fx:menuitem label="Edit">
                <fx:menuitem label="Cut" data="Cut"/>
                <fx:menuitem label="Copy" data="Copy"/>
                <fx:menuitem label="Paste" data="Paste"/>
            </fx:menuitem>
            <fx:menuitem label="View"/>
        </fx:XMLList>
        <fx:Array id="cmbDP">
            <fx:String>Item 1</fx:String>
            <fx:String>Item 2</fx:String>
            <fx:String>Item 3</fx:String>
        </fx:Array>
    </fx:Declarations>
    <mx:ApplicationControlBar id="dockedBar"
        dock="true">
        <mx:MenuBar height="100%"
            dataProvider="{menuXML}"
            labelField="@label"
            showRoot="true"/>
        <mx:HBox paddingBottom="5"
            paddingTop="5">
            <mx:ComboBox dataProvider="{cmbDP}"/>
            <mx:Spacer width="100%"/>
            <mx:TextInput id="myTI" text=""/>
            <mx:Button id="srch1"
                label="Search"
                click="Alert.show('Searching')"/>
        </mx:HBox>
    </mx:ApplicationControlBar>

    <mx:TextArea width="300" height="200"/>
</s:Application>
```

MX DividedBox, HDividedBox, and VDividedBox layout containers

The DividedBox layout container lays out its children horizontally or vertically, similar to a Box container, except that it inserts a divider between each child. You can use a mouse pointer to move the dividers in order to resize the area of the container allocated to each child. The DividedBox container can use MX and Spark components as children.

Use the `direction` property of a DividedBox container to determine vertical (default) or horizontal layout. The HDividedBox and VDividedBox containers are DividedBox containers with `horizontal` and `verticaldirection` property values.

The following example shows a DividedBox container:



Album	Artist	Comment	Price
Slanted and	Pavement	One of their	11.99
Brighten the	Pavement	My favorite.	11.99

One of their best. 4 Stars.

In this example, the outermost container is a horizontal DividedBox container. The horizontal divider marks the border between a Tree control and a vertical DividedBox container.

The vertical DividedBox container holds a DataGrid control (top) and a TextArea control (bottom). The vertical divider marks the border between these two controls.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a DividedBox, HDividedBox, or VDividedBox container

You use the `<mx:DividedBox>`, `<mx:VDividedBox>`, and `<mx:HDividedBox>` tags to define DividedBox containers. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. Typically, you use the VDividedBox (vertical DividedBox) and HDividedBox (horizontal DividedBox) containers as shortcuts so that you do not have to specify the `direction` property.

The following code example creates the image shown in "MX DividedBox, HDividedBox, and VDividedBox layout containers" on page 584:

```
<?xml version="1.0"?>
<!-- containers\layouts\HDivBoxSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    backgroundColor="white">
    <fx:Script>
      <![CDATA[
        private function myGrid_initialize():void {
          myGrid.dataProvider = [
            {Artist:'Pavement', Album:'Slanted and Enchanted',
                Price:11.99, Comment:'One of their best. 4 Stars.'},
            {Artist:'Pavement', Album:'Brighten the Corners',
                Price:11.99, Comment:'My favorite.'}
          ];
        }
      ]]>
    </fx:Script>
    <mx:HDividedBox width="100%" height="100%">
        <mx:Tree id="tree1"
            width="30%" height="100%"
            labelField="@label"
            showRoot="true">
            <fx:XMLList>
                <fx:menuitem label="Products">
                    <fx:menuitem label="Posters" isBranch="true"/>
                    <fx:menuitem label="CDs">
                        <fx:menuitem label="Pavement"/>
                        <fx:menuitem label="Pavarotti"/>
                        <fx:menuitem label="Phish"/>
                    </fx:menuitem>
                    <fx:menuitem label="T-shirts" isBranch="true"/>
                    <fx:menuitem label="Tickets" isBranch="true"/>
                </fx:menuitem>
            </fx:XMLList>
        </mx:Tree>
        <mx:VDividedBox width="70%" height="100%">
            <mx:DataGrid id="myGrid"
                width="100%" height="100%"
                initialize="myGrid_initialize();"
                change="currentMessage.text=
                    event.currentTarget.selectedItem.Comment;"/>
            <mx:TextArea id="currentMessage"
                width="100%"
                height="60"
                text="One of their best. 4 Stars."/>
        </mx:VDividedBox>
    </mx:HDividedBox>
</s:Application>
```

Notice that this example does not implement the logic to change the top area of the VDividedBox container when you select a node in the Tree control.

Using the dividers

The dividers of a DividedBox container let you resize the area of the container allocated for a child. However, for the dividers to function, the child has to be resizable; that is, it must specify a percentage-based size. So, a child with an explicit or default height or width cannot be resized in the corresponding direction by using a divider. Therefore, when you use the DividedBox container, you typically use percentage sizing for its children to make them resizable.

When you specify a percentage value for the `height` or `width` properties of a child to make it resizable, Flex initially sizes the child to the specified percentage, if possible. Then Flex can resize the child to take up all available space.

You can use the dividers to resize a percentage-sized child up to its maximum size, or down to its minimum size. To constrain the minimum size or maximum size of an area of the DividedBox, set an explicit value for the `minWidth` and `minHeight` properties or the `maxWidth` and `maxHeight` properties of the children in that area.

Using live dragging

By default, the DividedBox container disables live dragging. This means that the DividedBox container does not update the layout of its children until the user finishes dragging the divider, when the user releases the mouse button on a selected divider.

You can configure the DividedBox container to use live dragging by setting the `liveDragging` property to `true`. With live dragging enabled, the DividedBox container updates its layout as the user moves a divider. In some cases, you may encounter decreased performance if you enable live dragging.

MX Form, FormHeading, and FormItem layout containers

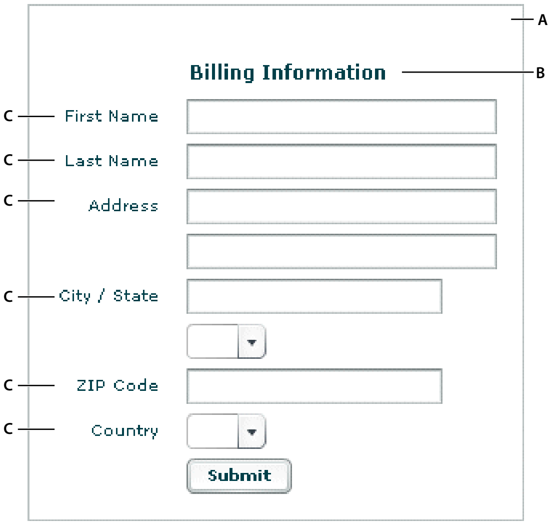
Forms are one of the most common methods that web applications use to collect information from users. Forms are used for collecting registration, purchase, and billing information, and for many other data collection tasks.

Note: *Adobe recommends that, when possible, you use the Spark Form container instead of the MX Form container. For more information, see "The Spark Form, Spark FormHeading, and Spark FormItem containers" on page 456.*

About forms

Flex supports form development by using the Form layout container and several child components of the Form container. The Form container lets you control the layout of a form, mark form fields as required or optional, handle error messages, and bind your form data to the Flex data model to perform data checking and validation. Also, you can apply style sheets to configure the appearance of your forms.

You use three different components to create your forms, as the following example shows:



A. *Form container* ***B.*** *FormHeading control* ***C.*** *FormItem containers*

For complete reference information, see Form, FormHeading, and FormItem in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating forms

You typically create a form by defining the following elements:

- The Form control
- FormHeading components, nested inside the Form control
- FormItem containers, nested inside the Form control
- Form fields, such as ComboBox and TextInput controls, nested inside the FormItem containers

  You can also include other components inside a form, such as HRule controls, as needed.

Creating the Form container

The Form container is the outermost container of a Flex form. The primary use of the Form container is to control the sizing and layout of the contents of the form, including the size of labels and the gap between items. The Form container always arranges its children vertically and left-aligns them in the form. The form container contains one or more FormHeading and FormItem containers.

You use the `<mx:Form>` tag to define the Form container. Specify an `id` value if you intend to refer to the entire form elsewhere in your MXML, either in another tag or in an ActionScript block.

The following code example shows the Form container definition for the form shown in the previous image in “About forms” on page 587:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Form id="myForm" width="400" height="100">
        <!-- Define FormHeading and FormItem components here -->
    </mx:Form>
</s:Application>
```

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FormHeading control

A FormHeading control specifies an optional label for a group of FormItem containers. The left side of the label is aligned with the left side of the controls in the form. You can have multiple FormHeading controls in your form to designate multiple content areas. You can also use FormHeading controls with a blank `label` to create vertical space in your form.

You use the `<mx:FormHeading>` tag to define a FormHeading container. Specify an `id` value if you intend to refer to the heading elsewhere in your MXML, either in another tag or in an ActionScript block.

The following code example defines the FormHeading control for the image shown in "About forms" on page 587:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormHeadingSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Form id="myForm" width="400" height="100">
        <mx:FormHeading label="Billing Information"/>
        <!--Define FormItem containers here. -->
    </mx:Form>
</s:Application>
```

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FormItem container

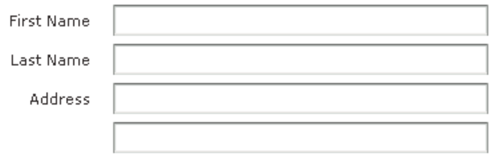
A FormItem container specifies a form element consisting of the following parts:

- A single label
- One or more child controls or containers, such as input controls

  The label is vertically aligned with the first child in the FormItem container and is right-aligned in the region to the left of the container.

  You use the `<mx:FormItem>` tag to define a FormItem container. Specify an `id` value if you intend to refer to the item elsewhere in your MXML, either in another tag or in an ActionScript block.

  Form containers typically contain multiple FormItem containers, as the following example shows:



In this example, you define three FormItem containers: one with the label First Name, one with the label Last Name, and one with the label Address. The Address FormItem container holds two controls to let a user enter two lines of address information. Each of the other two FormItem containers includes a single control.

For complete reference information, see FormItem in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Specifying form item direction

When you create a FormItem container, you specify its direction by using the value `vertical` (default) or `horizontal` for the `direction` property:

vertical Flex positions children vertically to the right of the FormItem label.

horizontal Flex positions children horizontally to the right of the FormItem label. If all children do not fit on a single row, they are divided into multiple rows with equal-sized columns. You can ensure that all children fit on a single line by using percentage-based widths or by specifying explicit widths wide enough for all of the components.

Controlling form item label style

You control the style of a FormItem label by setting the `labelStyleName` style property. The following example sets the FormItem label color to dark blue and its font size to 20 pixels:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormItemStyle.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        .myFormItemLabelStyle {
            color: #333399;
            fontSize: 20;
         }

    </fx:Style>
    <fx:Script>
        <![CDATA[
            private function processValues(zip:String, pn:String):void {
                // Validate and process data.
            }
        ]]>
    </fx:Script>
    <mx:Form id="myForm" defaultButton="{mySubmitButton}">
        <mx:FormItem label="Zip Code"
            labelStyleName="myFormItemLabelStyle">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit" id="mySubmitButton"
                click="processValues(zipCode.text, phoneNumber.text);"/>
        </mx:FormItem>

    </mx:Form>
</s:Application>
```

Example: A simple form

The following example shows the FormItem container definitions for the example form:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormComplete.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            private function submitForm():void {
                // Handle the form submission.
            }
        ]]>
    </fx:Script>
    <mx:Form id="myForm" width="400">
        <mx:FormHeading label="Billing Information"/>
        <mx:FormItem label="First Name">
            <mx:TextInput id="fname" width="100%"/>
        </mx:FormItem>
        <mx:FormItem label="Last Name">
            <mx:TextInput id="lname" width="100%"/>
        </mx:FormItem>
        <mx:FormItem label="Address">
            <mx:TextInput id="addr1" width="100%"/>
            <mx:TextInput id="addr2" width="100%"/>
        </mx:FormItem>
        <mx:FormItem label="City / State" direction="vertical">
            <mx:TextInput id="city"/>
            <mx:ComboBox id="st" width="75">
              <mx:ArrayList>
                 <fx:String>MA</fx:String>
                 <fx:String>NH</fx:String>
                 <fx:String>RI</fx:String>
              </mx:ArrayList>
            </mx:ComboBox>
        </mx:FormItem>
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zip" width="100"/>
        </mx:FormItem>
        <mx:FormItem label="Country">
            <mx:ComboBox id="cntry">
              <mx:ArrayList>
                 <fx:String>USA</fx:String>
                 <fx:String>UAE</fx:String>
                 <fx:String>UAW</fx:String>
              </mx:ArrayList>
            </mx:ComboBox>
        </mx:FormItem>
        <mx:FormItem>
            <mx:HRule width="200" height="1"/>
            <mx:Button label="Submit Form" click="submitForm();"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

Laying out forms

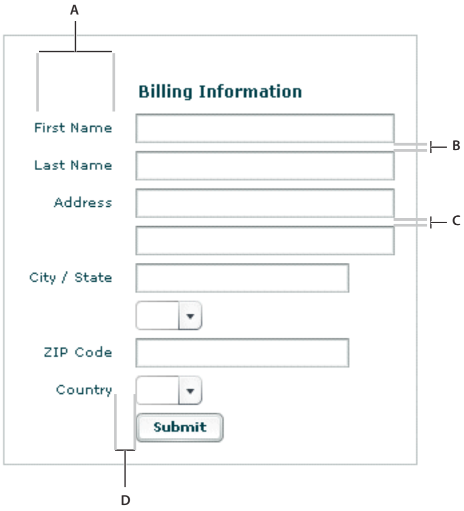
Flex determines the default size of a form in the following ways:

- The default height is large enough to hold the default or explicit heights of all the container children, plus the Form container top and bottom padding and the gaps between children.
- The default width is large enough to accommodate the widest FormItem label, plus the `indicatorGap` between the labels and the child controls, plus the widest default or explicit width among the child controls in the FormItems.

Aligning and spacing Form container children

All Form container labels are right-aligned, and all children are left-aligned in the container. You cannot override this alignment.

The following example shows the spacing of Form container children that you can control:



***A.** Form container: labelWidth **B.** Form container: verticalGap = 6 **C.** FormItem container: verticalGap = 6 **D.** Form container: indicatorGap = 14*

The following table describes the style properties that you use to control spacing and their default values:

Component	Style	Description	Default value
Form	`verticalGap`	Vertical space between Form container children	6 pixels
	`horizontalGap`	Horizontal space between Form container children	8 pixels
	`labelWidth`	Width of labels	Calculated by the container based on the child labels

	`paddingTop`
	`paddingBottom`
	`paddingLeft`
	`paddingRight`
		Border spacing around children	16 pixels on all sides

	`indicatorGap`	Gap between the end of the area in the form reserved for labels and the FormItem children or FormHeading heading	14 pixels

Component	Style	Description	Default value
FormHeading	`indicatorGap`	Overrides the indicator gap set by the `<mx:Form>` tag	14 pixels
	`paddingTop`	Gap between the top of the component and the label text	16 pixels
FormItem	`direction`	Direction of FormItem children: `vertical` or `horizontal`	`vertical`
	`horizontalGap`	Horizontal spacing between children in a FormItem container	8 pixels
	`labelWidth`	The width for the FormItem heading	The width of the label text

	`paddingTop`
	`paddingBottom`
	`paddingLeft`
	`paddingRight`
		Border spacing around the FormItem	0 pixels on all sides

	`verticalGap`	Vertical spacing between children in a FormItem container	6 pixels
	`indicatorGap`	Overrides the indicator gap set by the `<mx:Form>` tag	Determined by the `<mx:Form>` tag

Sizing and positioning Form container children

The Form layout container arranges children in a vertical column. The area of the Form container that is designated for children does not encompass the entire Form container. Instead, it starts at the right of the area defined by any labels and the gap defined by the `indicatorGap` property. For example, if the width of the Form container is 500 pixels, and the labels and `indicatorGap` property allocate 100 pixels of that width, the width of the child area is 400 pixels.

By default, Flex sizes the Form layout children vertically to their default height. Flex then determines the default width of each child, and stretches the child's width to the next highest quarter of the child area—that is, to one-quarter, one-half, three-quarters, or full width of the child area.

For example, if a container has a child area 400 pixels wide, and the default width of a TextArea control is 125 pixels, Flex stretches the TextArea control horizontally to the next higher quarter of the child area, the 200-pixel boundary, which is one-half of the child area. This sizing algorithm applies only to components without an explicitly specified width. It prevents your containers from having ragged right edges caused by controls with different widths.

You can also explicitly set the height or width of any control in the form to either a pixel value or a percentage of the Form size by using the `height` and `width` properties of the child.

Defining a default button

You use the `defaultButton` property of a container to define a default Button control. Pressing the Enter key while the focus is on any form control activates the Button control just as if it was explicitly selected.

For example, a login form displays user name and password inputs and a submit Button control. Typically, the user types a user name, tabs to the password field, types the password, and presses the Enter key to submit the login information without explicitly selecting the Button control. To define this type of interaction, set the `defaultButton` to the `id` of the submit Button control. In the following example, the event listener for the `click` event of submit button displays an Alert control, to show that Flex triggers this event if the user presses the Enter key when any form field has the focus. The commented-out line in the example would perform the more realistic action of invoking a web service to let the user log in.

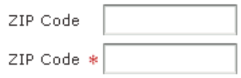
```
<?xml version="1.0"?>
<!-- containers\layouts\FormDefButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import flash.events.MouseEvent;
            import mx.controls.Alert;

            private function submitLogin(eventObj:MouseEvent):void {
                // Display an Alert to show the event happened.
                Alert.show("Login Requested");
                // Commented out to work without a web service.
                //myWebService.Login.send();
            }
        ]]>
    </fx:Script>
    <mx:Form defaultButton="{mySubmitButton}">
        <mx:FormItem label="Username">
            <mx:TextInput id="username"
                width="100"/>
        </mx:FormItem>
        <mx:FormItem label="Password">
            <mx:TextInput id="password"
                width="100"
                displayAsPassword="true"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button id="mySubmitButton"
                label="Login"
                click="submitLogin(event);"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

***Note:** When the drop-down list of a ComboBox control is open, pressing Enter selects the currently highlighted item in the ComboBox control; it does not activate the default button.*

Specifying required fields

Flex includes support for defining required input fields of a form. To define a required field, you specify the `required` property of the FormItem container. If this property is specified, all the children of the FormItem container are marked as required.

Flex inserts a red asterisk (*) character as a separator between the FormItem label and the FormItem child to indicate a required field. For example, the following example shows an optional ZIP code field and a required ZIP code field:



The following code example defines these fields:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormReqField.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Form>
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipOptional"
                width="100"/>
        </mx:FormItem>
        <mx:FormItem label="ZIP Code" required="true">
            <mx:TextInput id="zipRequired"
                width="100"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

You can enable the required indicator of a FormItem child at run time. This could be useful when the user input in one form field makes another field required. For example, you might have a form with a CheckBox control that the user selects to subscribe to a newsletter. Checking the box could make the user e-mail field required, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormReqFieldRuntime.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Form>
        <mx:FormItem label="Subscribe">
            <mx:CheckBox label="Subscribe?"
                click="emAddr.required=!emAddr.required;"/>
        </mx:FormItem>
        <mx:FormItem id="emAddr" label="E-mail Address">
            <mx:TextInput id="emailAddr"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

Flex does not perform any automatic enforcement of a required field; it only marks fields as required. You must add validation logic to your form to enforce it. As part of your enforcement logic, you can use Flex validators. All Flex validators have a `required` property, which is `true` by default. You can use validators in several ways, depending on how you enforce required fields and validation. For details, see “Validating Data” on page 1964.

Storing and validating form data

As part of designing your form, you must consider how you want to store your form data. In Flex, you have the following choices.

- Store the data within the form controls.
- Create a Flex data model to store your data.

Your decision about how to represent your data also affects how you perform input error detection or *data validation*, one of the primary tasks of a robust and stable form. You typically validate user input before you submit the data to the server. You can validate the user input within a submit function, or when a user enters data into the form.

Flex provides a set of data validators for the most common types of data collected by a form. You can use Flex validators with the following types of data:

- Credit card information
- Dates
- E-mail addresses
- Numbers
- Phone numbers
- Social Security Numbers
- Strings
- ZIP codes

As part of building your form, you can perform data validation by using your own custom logic, take advantage of the Flex data validation mechanism, or use a combination of custom logic and Flex data validation.

The following sections include information on how to initiate validation in a form; for detailed information on how to use Flex data validation, see “Validating Data” on page 1964.

Using Form controls to hold your form data

The following example uses Form controls to store the form data:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormData.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            private function processValues(zip:String, pn:String):void {
                // Validate and process data.
            }
        ]]>
    </fx:Script>
    <mx:Form id="myForm" defaultButton="{mySubmitButton}">
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit" id="mySubmitButton"
                click="processValues(zipCode.text, phoneNumber.text);"/>
        </mx:FormItem>

    </mx:Form>
</s:Application>
```

This example form defines two form controls: one for a ZIP code and one for a phone number. When you submit the form, you call a function that takes the two arguments that correspond to the data stored in each control. Your submit function can then perform any data validation on its inputs before processing the form data.

You don’t have to pass the data to the submit function. The submit function can access the form control data directly, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataSubmitNoArg.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = zipCode.text;
                var inputPhone:String = phoneNumber.text;
                // Check to see if pn is a number.
                // Check to see if zip is less than 4 digits.
                // Process data.
            }
        ]]>
    </fx:Script>
    <mx:Form id="myForm" defaultButton="{mySubmitButton}">
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit" id="mySubmitButton"
                click="processValues();"/>
        </mx:FormItem>

    </mx:Form>
</s:Application>
```

The technique of using the form fields directly, however, has the problem that the function is specific to the form and cannot easily be used by other forms.

Validating form control contents data on user entry

To validate form data upon user input, you can add Flex data validators to your application. The following example uses the ZipCodeValidator and PhoneNumberValidator to perform validation.

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataValidate.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = zipCode.text;
                var inputPhone:String = phoneNumber.text;
                // Perform any additional validation.
                // Process data.
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zcVal"
            source="{zipCode}" property="text"
            domain="US or Canada"/>
        <mx:PhoneNumberValidator id="pnVal"
            source="{phoneNumber}" property="text"/>
    </fx:Declarations>

    <mx:Form id="myForm" defaultButton="{mySubmitButton}">
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit" id="mySubmitButton"
                click="processValues();"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

If you validate the input data every time the user enters it, you might not have to do so again in your submit function. However, some validation in your submit function might still be necessary, especially if you want to ensure that two fields are valid when compared with each other.

For example, you can use Flex validators to validate a ZIP code field and state field individually. But you might want to validate that the ZIP code is valid for the specified state before submitting the form data. To do so, you perform a second validation in the submit function.

For detailed information on using validators, see “Validating Data” on page 1964.

Using a Flex data model to store form data

You can use a Flex data model to structure and store your form data and provide a framework for data validation. A data model stores data in fields that represent each part of a specific data set. For example, a *person* model might store information such as a person’s name, age, and phone number. You can then validate the data in the model based on the type of data stored in each model field.

The following example defines a Flex data model that contains two values. The two values correspond to the two input fields of a form.

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataModel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!-- Define the submit function that validates and
        processes the data. -->
    <fx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = myFormModel.zipCodeModel;
                var inputPhone:String = myFormModel.phoneNumberModel;
                // Process data.
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define data model. -->
        <fx:Model id="myFormModel">
            <info>
                <zipCodeModel>{zipCode.text}</zipCodeModel>
                <phoneNumberModel>{phoneNumber.text}</phoneNumberModel>
            </info>
        </fx:Model>
    </fx:Declarations>
    <!-- Define the form. -->
    <mx:Form borderStyle="solid">
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button id="b1"
                label="Submit"
                click="processValues();"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

You use the `<fx:Model>` tag to define the data model. Each child tag of the data model defines one field of the model. The tag body of each child tag in the model defines a *binding* to a form control. In this example, you bind the `zipCodeModel` model field to the text value of the `zipCode` TextInput control, and you bind the `phoneNumberModel` field to the text value of the `phoneNumber` TextInput control. For more information on data models, see “Storing data” on page 889.

When you bind a control to a data model, Flex automatically copies data from the control to the model upon user input. In this example, your submit function accesses the data from the model, not directly from the form controls.

Using Flex validators with form models

The following example modifies the example in the previous section by inserting two data validators—one for the ZIP code field and one for the phone number field:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataModelVal.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!-- Define the submit function that validates and processes the data -->
    <fx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = myFormModel.zipCodeModel;
                var inputPhone:String = myFormModel.phoneNumberModel;
                // Process data.
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define data model. -->
        <fx:Model id="myFormModel">
            <info>
                <zipCodeModel>{zipCode.text}</zipCodeModel>
                <phoneNumberModel>{phoneNumber.text}</phoneNumberModel>
            </info>
        </fx:Model>
        <!-- Define validators. -->
        <mx:ZipCodeValidator
            source="{myFormModel}" property="zipCodeModel"
            trigger="{zipCode}"
            listener="{zipCode}"/>
        <mx:PhoneNumberValidator
            source="{myFormModel}" property="phoneNumberModel"
            trigger="{b1}"
            listener="{phoneNumber}"
            triggerEvent="click"/>
    </fx:Declarations>
    <!-- Define the form. -->
    <mx:Form borderStyle="solid">
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button id="b1"
                label="Submit"
                click="processValues();"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

When the user enters data into the `zipCode` form field, Flex automatically copies that data to the data model. The `ZipCodeValidator` validator gets invoked when the user exits the `zipCode` form field, as specified by the validator's `trigger` property and the default value of the `triggerEvent` property, `valueCommit`. Flex then draws a red box around the `zipCode` field, as specified by the `listener` property.

When the user enters data into the `phoneNumber` form field, Flex automatically copies that data to the data model. The `PhoneNumberValidator` validator gets invoked when the user clicks the Button control, as specified by the validator's `trigger` and `triggerEvent` properties. Flex then draws a red box around the `phoneNumber` field, as specified by the `listener` property.

For detailed information on using validators, see "Validating Data" on page 1964.

Populating a Form control from a data model

Another use for data models is to include data in the model to populate form fields with values. The following example shows a form that reads static data from a data model to obtain the value for a form field:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataFromModel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <!-- Define data model. -->
        <fx:Model id="myFormModel">
            <info>
                <fName>{firstName.text}</fName>
                <lName>{lastName.text}</lName>
                <department>Accounting</department>
            </info>
        </fx:Model>
    </fx:Declarations>
    <mx:Form>
        <mx:FormItem label="First and Last Names">
            <mx:TextInput id="firstName"/>
            <mx:TextInput id="lastName"/>
        </mx:FormItem>
        <mx:FormItem label="Department">
            <mx:TextInput id="dept" text="{myFormModel.department}"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

This department data is considered static because the form always shows the same value in the field. You could also create a dynamic data model that takes the value of the department field from a web service, or calculates it based on user input.

For more information on data models, see "Storing data" on page 889.

Submitting data to a server

Form data is typically processed on a server, not locally on the client. Therefore, the submit event listener must have a mechanism for packing the form data for transfer to the server, and then handling any results returned from the server. In Flex, you typically use a web service, HTTP service, or remote Java object to pass data to the server.

You can also build logic into your submit function to control navigation of your application when the submit succeeds and when it fails. When the submit succeeds, you typically navigate to an area of your application that displays the results. If the submit fails, you can return control to the form so that the user can fix any errors.

The following example adds a web service to process form input data. In this example, the user enters a ZIP code, and then selects the Submit button. After performing any data validation, the submit event listener calls the web service to obtain the city name, current temperature, and forecast for the ZIP code.

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataSubmitServer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function processValues():void {
                // Check to see if ZIP code is valid.
                WeatherService.GetWeather.send();
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define the web service connection.
            The specified WSDL URI is not functional. -->
        <mx:WebService id="WeatherService"
            wsdl="/ws/WeatherService?wsdl">
            <mx:operation name="GetWeather">
                <mx:request>
                    <ZipCode>{zipCode.text}</ZipCode>
                </mx:request>
            </mx:operation>
        </mx:WebService>
    </fx:Declarations>
    <mx:Form>
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipCode"
                width="200"
                text="ZIP code please."/>
            <mx:Button
                width="60"
                label="Submit"
                click="processValues();"/>
        </mx:FormItem>
    </mx:Form>
    <mx:VBox>
        <mx:TextArea
            text=
              "{WeatherService.GetWeather.lastResult.CityShortName}"/>
        <mx:TextArea
            text=
              "{WeatherService.GetWeather.lastResult.CurrentTemp}"/>
        <mx:TextArea
            text=
          "{WeatherService.GetWeather.lastResult.DayForecast}"/>
    </mx:VBox>
</s:Application>
```

This example binds the form's input `zipCode` field directly to the `ZipCode` field of the web service request. To display the results from the web service, you bind the results to controls in a VBox container.

You have a great deal of flexibility when passing data to a web service. For example, you might modify this example to bind the input form field to a data model, and then bind the data model to the web service request. For more information on using web services, see Accessing server-side data.

You can also add event listeners for the web service to handle both a successful call to the web service, by using the `result` event, and a call that generates an error, by using the `fault` event. An error condition might cause you to display a message to the user with a description of the error. For a successful result, you might navigate to another section of your application.

The following example adds a `result` event and a `fault` event to the form. In this example, the form is defined as one child of a ViewStack container, and the form results are defined as a second child of the ViewStack container:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataSubmitServerEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function processValues():void {
                // Check to see if ZIP code is valid.
                WeatherService.GetWeather.send();
            }

            private function successfulCall():void {
                vs1.selectedIndex=1;
            }

            private function errorCall():void {
                Alert.show("Web service failed!", "Alert Box", Alert.OK);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define the web service connection.
            The specified WSDL URI is not functional. -->
        <mx:WebService id="WeatherService"
            wsdl="/ws/WeatherService?wsdl"
            result="successfulCall();"
            fault="errorCall();">
            <mx:operation name="GetWeather">
                <mx:request>
                    <ZipCode>{zipCode.text}</ZipCode>
                </mx:request>
            </mx:operation>
        </mx:WebService>
    </fx:Declarations>
    <mx:ViewStack id="vs1">
        <mx:Form>
            <mx:FormItem label="ZIP Code">
                <mx:TextInput id="zipCode"
                    width="200"
                    text="ZIP code please."/>
                <mx:Button width="60"
                    label="Submit"
                    click="processValues();"/>
            </mx:FormItem>
        </mx:Form>

        <mx:VBox>
            <mx:TextArea
                text=
                    "{WeatherService.GetWeather.lastResult.CityShortName}"/>
            <mx:TextArea
                text=
                    "{WeatherService.GetWeather.lastResult.CurrentTemp}"/>
            <mx:TextArea
                text=
                    "{WeatherService.GetWeather.lastResult.DayForecast}"/>
        </mx:VBox>
    </mx:ViewStack>
</s:Application>
```

When a call to the web service succeeds, the `successfulCall()` function switches the current ViewStack child to the VBox container to show the returned results. An error from the web service displays an Alert box, but does not change the current child of the ViewStack container; the form remains visible, which lets the user fix any input errors.

You have many options for handling navigation in your application based on the results of the submit function. The previous example used a ViewStack container to handle navigation. You might also choose to use a TabNavigator container or Accordion container for this same purpose.

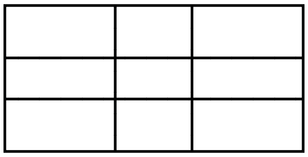
In some applications, you might choose to embed the form in a TitleWindow container. A TitleWindow container is a pop-up window that appears above the Adobe Flash Player drawing surface. In this scenario, users enter form data and submit the form from the TitleWindow container. If a submit succeeds, the TitleWindow container closes and displays the results in another area of your application. If a submit fails, Flex displays an error message and leaves the TitleWindow container visible.

Another type of application might use a dashboard layout, where you have multiple panels open on the dashboard. Submitting the form could cause another area of the dashboard to update with results, while a failure could display an error message.

For more information on the TabNavigator, Accordion, and TitleWindow containers, see "MX navigator containers" on page 628.

MX Grid layout container

You use a Grid layout container to arrange children as rows and columns of cells, much like an HTML table. The following example shows a Grid container that consists of nine cells arranged in a three-by-three pattern:



You can put zero or one child in each cell of a Grid container. If you include multiple children in a cell, put a container in the cell, and then add children to the container. The height of all cells in a row is the same, but each row can have a different height. The width of all cells in a column is the same, but each column can have a different width.

You can define a different number of cells for each row or each column of the Grid container. In addition, a cell can span multiple columns and/or multiple rows of the container.

If the default or explicit size of the child is larger than the size of an explicitly sized cell, the child is clipped at the cell boundaries.

If the child's default width or default height is smaller than the cell, the default horizontal alignment of the child in the cell is `left` and the default vertical alignment is `top`. You can use the `horizontalAlign` and `verticalAlign` properties of the `<mx:GridItem>` tag to control positioning of the child.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For information on the Tile container, which creates a layout where all cells have the same size, see "MX Tile layout container" on page 612.

Creating a Grid layout container

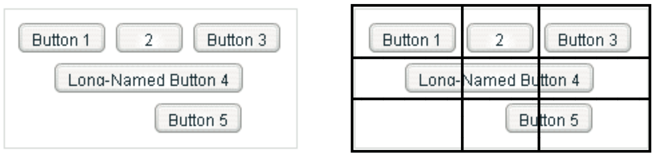
You create a Grid layout container as follows:

- You use the <mx:Grid> tag to define a Grid container; it can hold any number of `<mx:GridRow>` child tags.
- You use the <mx:GridRow> tag to define each row. It must be a child of the `<mx:Grid>` tag and can hold any number of `<mx:GridItem>` child tags.
- You use the <mx:GridItem> tag to define each row cell; it must be a child of the `<mx:GridRow>` tag.

The `<mx:GridItem>` tag takes the following optional properties that control how the item is laid out:

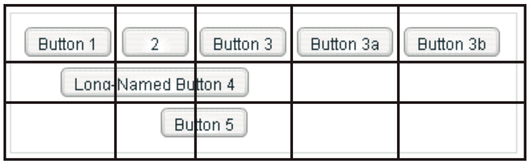
Property	Type	Use	Descriptions
`rowSpan`	Number	Property	Specifies the number of rows of the Grid container spanned by the cell. The default value is 1. You cannot extend a cell past the number of rows in the Grid container.
`colSpan`	Number	Property	Specifies the number of columns of the Grid container spanned by the cell. The default value is 1. You cannot extend a cell past the number of columns in the Grid container.

The following image shows a Grid container with three rows and three columns:



On the left, you see how the Grid container appears in Flash Player. On the right, you see the Grid container with borders overlaying it to illustrate the configuration of the rows and columns. In this example, the buttons in the first (top) row each occupy a single cell. The button in the second row spans three columns, and the button in the third row spans the second and third columns.

You do not have to define the same number of cells for every row of a Grid container, as the following image shows. The Grid container defines five cells in row one; row two has one item that spans three cells; and row 3 has one empty cell, followed by an item that spans two cells.



The following MXML code creates the Grid container with three rows and three columns shown in the first image in this section:

```
<?xml version="1.0"?>
<!-- containers\layouts\GridSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Grid id="myGrid">
        <!-- Define Row 1. -->
        <mx:GridRow id="row1">
            <!-- Define the first cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 1"/>
            </mx:GridItem>
            <!-- Define the second cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="2"/>
            </mx:GridItem>
            <!-- Define the third cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 3"/>
            </mx:GridItem>
        </mx:GridRow>
        <!-- Define Row 2. -->
        <mx:GridRow id="row2">
            <!-- Define a single cell to span three columns of Row 2. -->
            <mx:GridItem colSpan="3" horizontalAlign="center">
                <mx:Button label="Long-Named Button 4"/>
            </mx:GridItem>
        </mx:GridRow>
        <!-- Define Row 3. -->
        <mx:GridRow id="row3">
            <!-- Define an empty first cell of Row 3. -->
            <mx:GridItem/>
            <!-- Define a cell to span columns 2 and 3 of Row 3. -->
            <mx:GridItem colSpan="2" horizontalAlign="center">
                <mx:Button label="Button 5"/>
            </mx:GridItem>
        </mx:GridRow>
    </mx:Grid>
</s:Application>
```

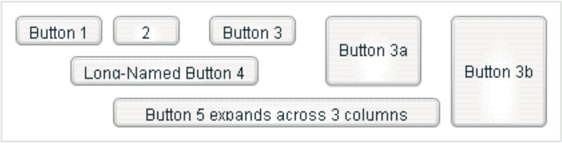
To modify the preceding example to include five buttons across the top row, you modify the first `<mx:GridRow>` tag as follows:

```
<?xml version="1.0"?>
<!-- containers\layouts\Grid5Button.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Grid id="myGrid">
        <!-- Define Row 1. -->
        <mx:GridRow id="row1">
            <!-- Define the first cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 1"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="2"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3a"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3b"/>
            </mx:GridItem>
        </mx:GridRow>
        <!-- Define Row 2. -->
        <mx:GridRow id="row2">
            <!-- Define a single cell to span three columns of Row 2. -->
            <mx:GridItem colSpan="3" horizontalAlign="center">
                <mx:Button label="Long-Named Button 4"/>
            </mx:GridItem>
        </mx:GridRow>
        <!-- Define Row 3. -->
        <mx:GridRow id="row3">
            <!-- Define an empty first cell of Row 3. -->
            <mx:GridItem/>
            <!-- Define a cell to span columns 2 and 3 of Row 3. -->
            <mx:GridItem colSpan="2" horizontalAlign="center">
                <mx:Button label="Button 5"/>
            </mx:GridItem>
        </mx:GridRow>
    </mx:Grid>
</s:Application>
```

Setting the row and column span

The `colSpan` and `rowSpan` properties of the GridItem container let you create grid items that occupy multiple grid rows and columns. Making an item span multiple rows or columns does not necessarily make its child control or container larger; you must size the child so it fits the space appropriately, as the following example shows.

The following image shows a modification to the example in Creating a Grid layout container, where Button 3a now spans two rows, Button 3b spans three rows, and Button 5 spans three columns:



The following code shows the changes that were made to produce these results:

```
<?xml version="1.0"?>
<!-- containers\layouts\GridRowSpan.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Grid id="myGrid">
        <!-- Define Row 1. -->
        <mx:GridRow id="row1" height="33%">
            <!-- Define the first cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 1"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="2"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3"/>
            </mx:GridItem>
            <mx:GridItem rowSpan="2">
                <mx:Button label="Button 3a" height="100%"/>
            </mx:GridItem>
            <mx:GridItem rowSpan="3">
                <mx:Button label="Button 3b" height="100%"/>
            </mx:GridItem>
        </mx:GridRow>
        <!-- Define Row 2. -->
        <mx:GridRow id="row2" height="33%">
            <!-- Define a single cell to span three columns of Row 2. -->
            <mx:GridItem colSpan="3" horizontalAlign="center">
                <mx:Button label="Long-Named Button 4"/>
            </mx:GridItem>
        </mx:GridRow>
        <!-- Define Row 3. -->
        <mx:GridRow id="row3" height="33%">
            <!-- Define an empty first cell of Row 3. -->
            <mx:GridItem/>
            <!-- Define a cell to span columns 2 and 3 and 4 of Row 3. -->
            <mx:GridItem colSpan="3">
                <mx:Button
                    label="Button 5 expands across 3 columns"
                    width="75%"/>
            </mx:GridItem>
        </mx:GridRow>
    </mx:Grid>
</s:Application>
```

This example makes several changes with the following effects:

- It sets the height of each row to 33% of the grid, ensuring that the rows have equal heights.
- It sets the `rowSpan` properties of the items with Buttons 3a and 3b to make them span two and three rows, respectively.
- It sets the `height` properties of Buttons 3a and 3b to 100% to make these buttons fill all rows that they span. If you omit this property on these buttons, Flex sets the buttons to their default height, so they do not appear to span the rows.
- It sets Button 5 to span three rows and sets a percentage-based width of 75%. In this example, the text requires the button to fill the available width of the three columns, so Flex sizes the button to the default size that fits the text, not the requested 75%. If you omit the `width` property, the result is identical. To see the effect of the percentage width specification, keep the specification and shorten the label text; the button then spans three-quarters of the three columns, centered on the middle column.

  The resulting grid has the several additional characteristics. Although the second row definition specifies only a single `<mx:GridItem>` tag that defines a cell spanning three columns, Flex automatically creates two additional cells to allow Buttons 3a and 3b to expand into the row. The third row also has five cells. The first cell is defined by the empty `<mx:gridItem/>` tag. The second through fourth cells are defined by the `GridItem` that contains Button 5, which spans three columns. The fifth column is created because the last item in the first row spans all three rows.

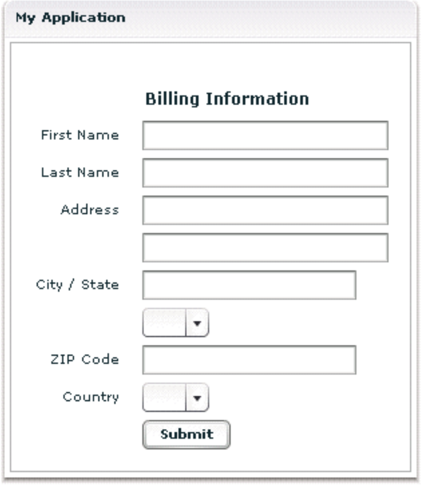
MX Panel layout container

A Panel layout container includes a title bar, a title, a status message area (in the title bar), a border, and a content area for its children. Typically, you use Panel containers to wrap self-contained application modules. For example, you could define several Panel containers in your application where one Panel container holds a form, a second holds a shopping cart, and a third holds a catalog.

***Note:** Adobe recommends that, when possible, you use the Spark Panel container instead of the MX Panel container. For more information, see "The Spark Panel container" on page 434.*

The Panel container has a `layout` property that lets you specify one of three types of layout: `horizontal`, `vertical` (default), or `absolute` layout. Horizontal and vertical layout use the Flex automatic layout rules to lay out children horizontally or vertically. Absolute layout requires you to specify each child's *x* and *y* position relative to the panel contents area, or to use constraint-based layout styles to anchor one or more sides or the container horizontal or vertical center relative to the panel content area. For examples of using absolute and constraint-based layout in a container, see "Creating and using a Canvas control" on page 574. For detailed information on using these layout techniques, see "Laying out components" on page 359.

The following example shows a Panel container with a Form container as its child:



For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a Panel layout container

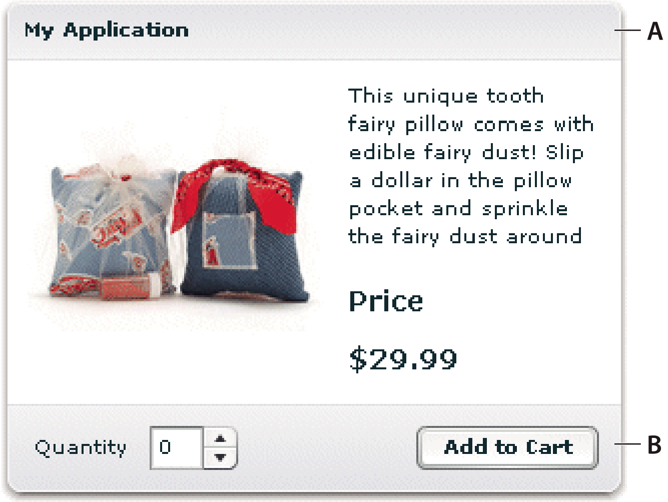
You define a Panel container in MXML by using the `<mx:Panel>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example defines a Panel container that contains a form as the top-level container in your application. In this example, the Panel container provides you with a mechanism for including a title bar, as in a standard GUI window.

```
<?xml version="1.0"?>
<!-- containers\layouts\PanelSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Panel id="myPanel" title="My Application">
        <mx:Form width="300">
            <mx:FormHeading label="Billing Information"/>
            <!-- Form contents goes here -->
        </mx:Form>
    </mx:Panel>
</s:Application>
```

Adding a ControlBar container to a Panel container

You can use the ControlBar container with a Panel container to hold components that can be shared by the other children in the Panel container. The RichTextEditor control, for example, consists of a Panel control with a TextArea control and a custom ControlBar for the text formatting controls. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart, as the following example shows:


My Application
This unique tooth fairy pillow comes with edible fairy dust! Slip a dollar in the pillow pocket and sprinkle the fairy dust around
Price
$29.99
Quantity
0
Add to Cart
A
B

A. Panel container B. ControlBar container

You specify the `<mx:ControlBar>` tag as the last child tag of an `<mx:Panel>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\PanelCBar.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Panel title="My Application"
        paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10"
        width="300">

        <fx:Script>
            <![CDATA[
                private function addToCart():void {
                    // Handle event.
                }
            ]]>
        </fx:Script>

        <mx:HBox width="100%">
            <!-- Area for your catalog. -->
        </mx:HBox>

        <mx:ControlBar width="100%">
            <mx:Label text="Quantity"/>
            <mx:NumericStepper id="myNS"/>
            <!-- Use Spacer to push Button control to the right. -->
            <mx:Spacer width="100%"/>
            <mx:Button label="Add to Cart" click="addToCart();"/>
        </mx:ControlBar>
    </mx:Panel>
</s:Application>
```

For more information on the ControlBar container, see “MX ControlBar layout container” on page 581.

You can also add the ControlBar container dynamically to a Panel container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\PanelCBarDynamicAdd.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            import mx.containers.ControlBar;
            import mx.controls.*;
            import flash.events.MouseEvent;

            private var myCB:ControlBar=new ControlBar();
            private var myLabel:Label=new Label();
            private var myNS:NumericStepper=new NumericStepper();
            private var mySpacer:Spacer=new Spacer();
            private var myButton:Button=new Button();

            private var canAddChild:Boolean = true;

            private function addCBHandler():void {
                if (canAddChild) {
                    /* Create Controlbar control. */
                    myLabel.text="Quantity";
                    mySpacer.percentWidth=100;
                    myButton.label="Add to Cart";
                    myButton.addEventListener('click', addToCart);

                    myCB.percentWidth=100;
                    myCB.addChild(myLabel);
                    myCB.addChild(myNS);
                    myCB.addChild(mySpacer);
                    myCB.addChild(myButton);

                    /* Add the ControlBar as the last child of the
                       Panel container.
                       The ControlBar appears in the normal content area
                       of the Panel container. */
                    myPanel.addChildAt(myCB, myPanel.numChildren);

                    /* Call createComponentsFromDescriptors() to make the
                       ControlBar appear in the bottom border area
                       of the Panel container. The ControlBar must be the
                       last child in the Panel container. */
                    myPanel.createComponentsFromDescriptors();

                    /* Prevents more than one ControlBar control from being added. */
                    canAddChild = false;
                }
            }
            private function addToCart(event:MouseEvent):void {
                Alert.show("ControlBar Button clicked.");
            }
        ]]>
    </fx:Script>

    <mx:Panel id="myPanel"
        title="My Application"
        paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10"
        width="300">

        <mx:HBox width="100%">
            <!-- Area for your catalog. -->
        </mx:HBox>
        <mx:Button label="Add ControlBar"
            click="addCBHandler();"/>
    </mx:Panel>
</s:Application>
```

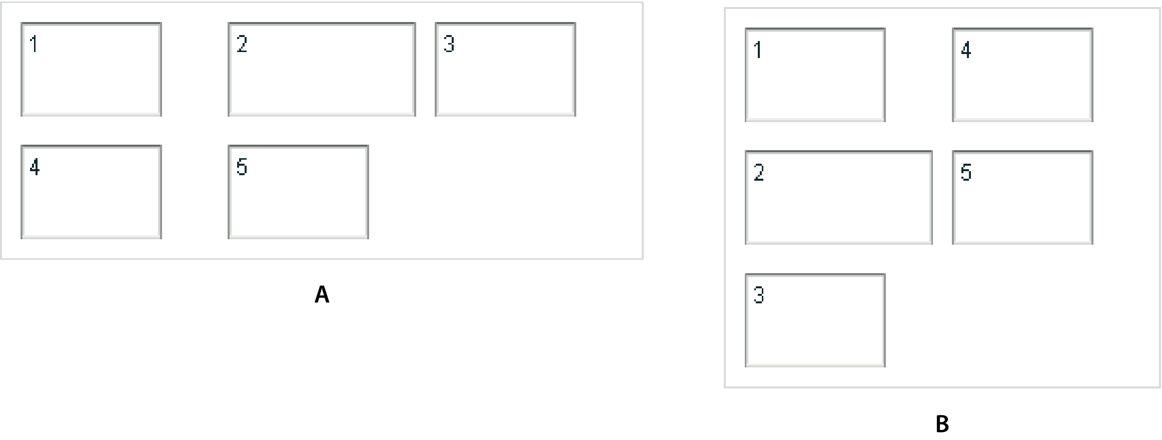
MX Tile layout container

A Tile layout container lays out its children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The `direction` property determines the layout. The valid values for the `direction` property are `vertical` for a column layout and `horizontal` (default) for a row layout.

***Note:** Adobe recommends that, when possible, you use a Spark container with TileLayout instead of the MX Tile container. For more information, see "About Spark layouts" on page 420.*

All Tile container cells have the same size, unlike the cells of a Grid layout container (see “MX Grid layout container” on page 603). Flex arranges the cells of a Tile container in a square grid, where each cell holds a single child component. For example, if you define 16 children in a Tile layout container, Flex lays it out four cells wide and four cells high. If you define 13 children, Flex still lays it out four cells wide and four cells high, but leaves the last three cells in the fourth row empty.

The following image shows examples of horizontal and vertical Tile containers:



***A.** Horizontal (default)* ***B.** Vertical*

For complete reference information, see Tile in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a Tile layout container

You define a Tile container in MXML by using the `<mx:Tile>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. You can use the `tileHeight` and `tileWidth` properties to specify explicit tile dimensions.

The following example creates the horizontal Tile container shown in the image in “MX Tile layout container” on page 612. All cells have the height and width of the largest child, 50 pixels high and 100 pixels wide.

```
<?xml version="1.0"?>
<!-- containers\layouts\TileSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Tile id="myFlow"
        direction="horizontal"
        borderStyle="solid"
        paddingTop="10" paddingBottom="10"
        paddingRight="10" paddingLeft="10"
        verticalGap="15" horizontalGap="10">

        <mx:TextInput id="text1" text="1" height="50" width="75"/>
        <mx:TextInput id="text2" text="2" height="50" width="100"/>
        <mx:TextInput id="text3" text="3" height="50" width="75"/>
        <mx:TextInput id="text4" text="4" height="50" width="75"/>
        <mx:TextInput id="text5" text="5" height="50" width="75"/>
    </mx:Tile>
</s:Application>
```

Sizing and positioning a child in a Tile container

Flex sets the default size of each Tile cell to the height of the tallest child and the width of the widest child. All cells have the same default size. If the default size of a child is larger than the cell because, for example, you used the `tileHeight` and `tileWidth` properties to explicitly size the cells, Flex automatically sizes the child to fit inside the cell boundaries. This, in turn, may clip the content inside the control; for instance, the label of a button might be clipped even though the button itself fits into the cell. If you specify an explicit child dimension that is greater than the `tileHeight` or `tileWidth` property, Flex clips the child.

If the child's default width or default height is smaller than the cell, the default horizontal alignment of the child in the cell is `left` and the default vertical alignment is `top`. You can use the `horizontalAlign` and `verticalAlign` properties of the `<mx:Tile>` tag to control the positioning of the child.

If the child uses percentage-based sizing, the child is enlarged or shrunk to the specified percentage of the cell. In the example in "Creating a Tile layout container" on page 613, the TextInput control named `text2` has a width of 100 pixels; therefore, the default width of all Tile cells is 100 pixels, so most children are smaller than the cell size. If you want all child controls to increase in size to the full width of the cells, set the `width` property of each child to 100%, as the following example shows. The example also shows how to use the Tile control's `tileWidth` property to specify the width of the tiles:

```
<?xml version="1.0"?>
<!-- containers\layouts\TileSizing.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Tile id="myFlow"
        direction="horizontal"
        borderStyle="solid"
        paddingTop="10" paddingBottom="10"
        paddingRight="10" paddingLeft="10"
        verticalGap="15" horizontalGap="10"
        tileWidth="100">

        <mx:TextInput id="fname" text="1" height="50" width="100%"/>
        <mx:TextInput id="lname" text="2" height="50" width="100%"/>
        <mx:TextInput id="addr1" text="3" height="50" width="100%"/>
        <mx:TextInput id="addr2" text="4" height="50" width="100%"/>
        <mx:TextInput id="addr3" text="5" height="50" width="100%"/>
    </mx:Tile>
</s:Application>
```

***Note:** When you set the child's `width` and `height` properties to percentages, the percentage is based on the size of the tile cell, not on the size of the Tile container itself. Even though it isn't an explicitly defined container, the cell acts as the parent of the components in the Tile container.*

MX TitleWindow layout container

A TitleWindow layout container is a Panel container that is optimized for use as a pop-up window. The container consists of a title bar, a caption and status area in the title bar, a border, and a content area for its children. Unlike the Panel container, it can display a Close button, and is designed to work as a pop-up window that users can drag around the screen the application window.

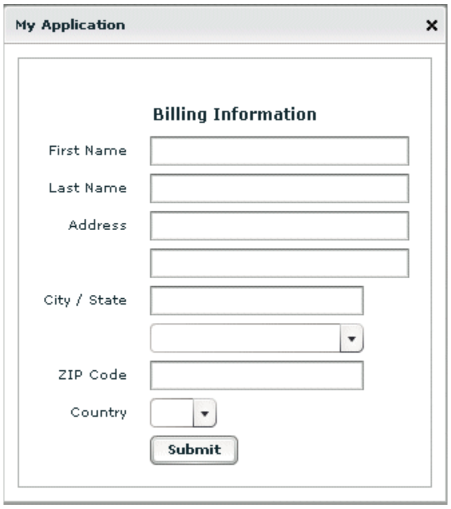
***Note:** Adobe recommends that, when possible, you use the Spark TitleWindow container instead of the MX TitleWindow container. For more information, see "The Spark TitleWindow container" on page 443.*

A pop-up TitleWindow container can be *modal*, which means that it takes all keyboard and mouse input until it is closed, or *nonmodal*, which means other windows can accept input while the pop-up window is still open.

One typical use for a TitleWindow container is to hold a form. When the user completes the form, you can close the TitleWindow container programmatically, or let the user request the application to close it by using the close icon (a box with an *x* inside it) in the upper-right corner of the window.

Because you pop up a Title window, you do not create it directly in MXML, as you do most controls; instead you use the PopUpManager.

The following example shows a TitleWindow container with a Form container as its child:



For complete reference information, see TitleWindow in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Using the PopUpManager to create a TitleWindow container

To create and remove a pop-up TitleWindow container, you use methods of the PopUpManager. The PopUpManager is in the mx.managers package.

Creating a pop-up window

To create a pop-up window, use the PopUpManager`createPopUp()` method. The `createPopUp()` method has the following signature:

```
public static createPopUp(parent:DisplayObject, class:Class,
    modal:Boolean = false):IFlexDisplayObject
```

The method has the following arguments.

Argument	Description
`parent`	A reference to a window to pop-up over.
`class`	A reference to the class of object you want to create, typically a custom MXML component that implements a TitleWindow container.
`modal`	(Optional) A Boolean value that indicates whether the window is modal, and takes all mouse input until it is closed (`true`), or whether interaction is allowed with other controls while the window is displayed (`false`). The default value is `false`.

Note: *Flex continues executing code in the parent even after you create a modal pop-up window.*

You can also create a pop-up window by passing an instance of a TitleWindow class or custom component to the PopUpManager `addPopUp()` method. For more information, see “Using the addPopUp() method” on page 626.

Defining a custom TitleWindow component

One of the most common ways of creating a TitleWindow container is to define it as a custom MXML component.

- You define the TitleWindow container, its event handlers, and all of its children in the custom component.
- You use the PopUpManager `createPopUp()` and `removePopUp()` methods to create and remove the TitleWindow container.

The following example code defines a custom MyLoginForm TitleWindow component that is used as a pop-up window:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\MyLoginForm.mxml -->
<mx:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;

            private function processLogin():void {
                // Check credentials (not shown) then remove pop up.
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
    <mx:Form>
        <mx:FormItem label="User Name">
            <mx:TextInput id="username" width="100%"/>
        </mx:FormItem>
        <mx:FormItem label="Password">
            <mx:TextInput id="password"
                displayAsPassword="true"
                width="100%"/>
        </mx:FormItem>
    </mx:Form>
    <mx:HBox>
        <mx:Button click="processLogin();" label="OK"/>
        <mx:Button
            label="Cancel"
            click="PopUpManager.removePopUp(this);"/>
    </mx:HBox>
</mx:TitleWindow>
```

This file, named MyLoginForm.mxml, defines a TitleWindow container by using the `<mx:TitleWindow>` tag. The TitleWindow container defines two TextInput controls, for user name and password, and two Button controls, for submitting the form and for closing the TitleWindow container. This example does not include the code for verifying the user name and password in the `submitForm()` event listener.

In this example, you process the form data in an event listener of the MyLoginForm.mxml component. To make this component more reusable, you can define the event listeners in your main application. This lets you create a generic form that leaves the data handling to the application that uses the form. For an example that defines the event listeners in the main application, see “Using TitleWindow and PopUpManager events” on page 620.

Using the PopUpManager to create the pop-up TitleWindow

To create a pop-up window, you call the PopUpManager `createPopUp()` method and pass it the parent, the name of the class that creates the pop-up, and the modal Boolean value. The following main application code creates the TitleWindow container defined in Defining a custom TitleWindow component:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginForm.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import mx.core.IFlexDisplayObject;
            import myComponents.MyLoginForm;
            private function showLogin():void {
                // Create a non-modal TitleWindow container.
                var helpWindow:IFlexDisplayObject =
                    PopUpManager.createPopUp(this, MyLoginForm, false);
            }
        ]]>
    </fx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
    </mx:VBox>
</s:Application>
```

In this example, when the user selects the Login button, the event listener for the `click` event uses the `createPopUp()` method to create a TitleWindow container, passing to it the name of the MyLoginForm.mxml file as the class name.

Often, you cast the return value of the `createPopUp()` method to TitleWindow so that you can manipulate the properties of the pop-up TitleWindow container, as the following version of the `showLogin()` method from the preceding example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginFormCast.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
          import mx.managers.PopUpManager;
          import mx.core.IFlexDisplayObject;
          import myComponents.MyLoginForm;
          // Additional import statement to use the TitleWindow container.
          import mx.containers.TitleWindow;
          private function showLogin():void {
            // Create the TitleWindow container.
            var helpWindow:TitleWindow =
          TitleWindow(PopUpManager.createPopUp(this, MyLoginForm, false));
            // Add title to the title bar.
            helpWindow.title="Enter Login Information";
            // Make title bar slightly transparent.
            helpWindow.setStyle("borderAlpha", 0.9);
            // Add a close button.
            // To close the container, your must also handle the close event.
            helpWindow.showCloseButton=true;
          }
        ]]>
    </fx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
    </mx:VBox>
</s:Application>
```

Removing a pop-up window

To remove a pop-up TitleWindow, use the PopUpManager `removePopUp()` method. You pass the object created with the `PopUpManager.createPopUp()` method to the `removePopUp()` method. The following modification to the example from "Using the PopUpManager to create the pop-up TitleWindow" on page 617removes the pop-up when the user clicks the Done button:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginFormRemove.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
          import mx.managers.PopUpManager;
          import myComponents.MyLoginForm;
          import mx.core.IFlexDisplayObject;
          private var helpWindow:IFlexDisplayObject;
          private function showLogin():void {
            // Create the TitleWindow container.
            helpWindow = PopUpManager.createPopUp(this, MyLoginForm, false);
          }
          private function removeForm():void {
            PopUpManager.removePopUp(helpWindow);
          }
        ]]>
    </fx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
        <mx:Button id="b2" label="Remove Form" click="removeForm();"/>
    </mx:VBox>
</s:Application>
```

You commonly call the `removePopUp()` method from a TitleWindow `close` event and the PopUpManager `mouseDownOutside` event, as described in “Passing data to and from a pop-up window” on page 622.

Centering a pop-up window

Call the `centerPopUp()` method of the PopUpManager to center the pop-up within another container. The following custom MXML component centers itself in its parent container:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\MyLoginFormCenter.mxml -->
<mx:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="handleCreationComplete();">

    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;

            private function handleCreationComplete():void {
                // Center the TitleWindow container
                // over the control that created it.
                PopUpManager.centerPopUp(this);
            }

            private function processLogin():void {
                // Check credentials (not shown) then remove pop up.
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
    <mx:Form>
        <mx:FormItem label="User Name">
            <mx:TextInput id="username" width="100%"/>
        </mx:FormItem>
        <mx:FormItem label="Password">
            <mx:TextInput id="password"
                width="100%"
                displayAsPassword="true"/>
        </mx:FormItem>
    </mx:Form>
    <mx:HBox>
        <mx:Button click="processLogin();" label="OK"/>
        <mx:Button
            label="Cancel"
            click="PopUpManager.removePopUp(this);"/>
    </mx:HBox>
</mx:TitleWindow>
```

Using TitleWindow and PopUpManager events

You can add a close icon (a small *x* in the upper-right corner of the TitleWindow title bar) to make it appear similar to dialog boxes in a GUI environment. You do this by setting the `showCloseButton` property to `true`, as the following example shows:

```
<mx:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    showCloseButton="true">
```

The default value for `showCloseButton` is `false`.

The TitleWindow broadcasts a `close` event when the user clicks the close icon. You must create a handler for that event and close the TitleWindow from within that event handler. Flex does not close the window automatically.

In the simplest case, you can call the PopUpManager `removePopUp()` method directly in the TitleWindow `close` event property specify, as the following line shows:

```
<mx:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    "showCloseButton="true"
    close="PopUpManager.removePopUp(this);">
```

If you need to clean up, before closing the TitleWindow control, create an event listener function for the close event and close the TitleWindow from within that handler, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\MyLoginFormRemoveMe.mxml -->
<mx:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    showCloseButton="true"
    close="removeMe();">
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            private function removeMe():void {
                // Put any clean-up code here.
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
</mx:TitleWindow>
```

You can also use the PopUpManager `mouseDownOutside` event to close the pop-up window when a user clicks the mouse outside the TitleWindow. To do this, you add an event listener to the TitleWindow instance for the `mouseDownOutside` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginFormMouseDown.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import mx.containers.TitleWindow;
            import myComponents.MyLoginForm;
            import mx.events.FlexMouseEvent;
            private var helpWindow:TitleWindow;
            private function showLogin():void {
                // Create the TitleWindow container.
                helpWindow = TitleWindow(PopUpManager.createPopUp(this,
                    MyLoginForm, false));
                helpWindow.addEventListener("mouseDownOutside", removeForm);
            }
            private function removeForm(event:FlexMouseEvent):void {
                PopUpManager.removePopUp(helpWindow);
            }
        ]]>
    </fx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
    </mx:VBox>
</s:Application>
```

You define the event listener in the main application, and then assign it to the pop-up window when you create it. This technique lets you use a generic pop-up window, defined by the component MyLoginForm.mxml, and then modify the behavior of the component by assigning event listeners to it from the main application.

Creating a modal pop-up window

The `createPopUp()` method takes an optional *modal* parameter. You can set this parameter to `true` to make the window modal. When a TitleWindow is modal, you cannot select any other component while the window is open. The default value of *modal* is `false`.

The following example creates a modal pop-up window:

```
var pop1:IFlexDisplayObject = PopUpManager.createPopUp(this, MyLoginForm,
    true);
```

Passing data to and from a pop-up window

To make your pop-up window more flexible, you might want to pass data to it or return data from it. To do this, use the following guidelines:

- Create a custom component to be your pop-up. In most circumstances, this component is a TitleWindow container.
- Declare variables in your pop-up that you will set in the application that creates the pop-up.
- Cast the pop-up to be the same type as your custom component.
- Pass a reference to that component to the pop-up window, if the pop-up window is to set a value on the application or one of the application's components.

For example, the following application populates a ComboBox in the pop-up window with an Array defined in the main application.

When creating the pop-up, the application casts the pop-up to be of type ArrayEntryForm, which is the name of the custom component that defines the pop-up window. If you do not do this, the application cannot access the properties that you create.

The application passes a reference to the TextInput component in the Application container to the pop-up window so that the pop-up can write its results back to the container. The application also passes the Array of file extensions for the pop-up ComboBox control's data provider, and sets the pop-up window's title. By setting these in the application, you can reuse the pop-up window in other parts of the application without modification, because it does not have to know the name of the component it is writing back to or the data that it is displaying, only that its data is in an Array and it is writing to a TextArea.

```
<?xml version="1.0"?>
<!-- containers\layouts\MainArrayEntryForm.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import myComponents.ArrayEntryForm;

            public var helpWindow:Object;

            public function displayForm():void {
                /* Array with data for the custom control ComboBox control. */
                var doctypes:Array = ["*.as", "*.mxml", "*.swc"]

                /* Create the pop-up and cast the
```

```
                  return value of the createPopUp()
                  method to the ArrayEntryForm custom component. */
               var pop1:ArrayEntryForm = ArrayEntryForm(
                   PopUpManager.createPopUp(this, ArrayEntryForm, true));

               /* Set TitleWindow properties. */
               pop1.title="Select File Type";
               pop1.showCloseButton=true;

               /* Set properties of the ArrayEntryForm custom component. */
               pop1.targetComponent = ti1;
               pop1.myArray = doctypes;
               PopUpManager.centerPopUp(pop1);
           }
       ]]>
    </fx:Script>
    <mx:TextInput id="ti1" text=""/>
    <mx:Button id="b1" label="Select File Type"
        click="displayForm();"/>
</s:Application>
```

The following custom component, ArrayEntryForm.mxml, declares two variables. The first one is for the Array that the parent application passes to the pop-up window. The second holds a reference to the parent application's TextInput control. The component uses that reference to update the parent application:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\ArrayEntryForm.mxml -->
<mx:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
        showCloseButton="true"
        width="200" borderAlpha="1"
        close="removeMe();">

    <fx:Script>
        <![CDATA[
            import mx.controls.TextInput;
            import mx.managers.PopUpManager;

            // Variables whose values are set by the main application.
            // Data provider array for the component's ComboBox control.
            [Bindable]
            public var myArray:Array;
            // A reference to the TextInput control
            // in which to put the result.
            public var targetComponent:TextInput;
            // OK button click event listener.
            // Sets the target component in the application to the
            // selected ComboBox item value.
            private function submitData():void {
                targetComponent.text = String(cb1.selectedItem);
                removeMe();
            }
            // Cancel button click event listener.
            private function removeMe():void {
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
    <mx:ComboBox id="cb1" dataProvider="{myArray}"/>
    <mx:HBox>
        <mx:Button label="OK" click="submitData();"/>
        <mx:Button label="Cancel" click="removeMe();"/>
    </mx:HBox>
</mx:TitleWindow>
```

From within a pop-up custom component, you can also access properties of the parent application by using the `parentApplication` property. For example, if the application has a Button control named b1, you can get the label of that Button control, as the following example shows:

```
myLabel = parentApplication.b1.label;
```

This technique, however, uses a hard-coded value in the pop-up component for both the target component id in the parent and the property in the component.

Passing data using events

The following example modifies the example from the previous section to use event listeners defined in the main application to handle the passing of data back from the pop-up window to the main application. This example shows the ArrayEntryFormEvents.mxml file with no event listeners defined within it.

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\ArrayEntryFormEvents.mxml -->
<mx:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    showCloseButton="true"
    width="200"
    borderAlpha="1">

    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;

            // Variables whose values are set by the main application.
            // Data provider array for the component's ComboBox control.
            [Bindable]
            public var myArray:Array;
        ]]>
    </fx:Script>
    <mx:ComboBox id="cb1" dataProvider="{myArray}"/>
    <mx:HBox>
        <mx:Button id="okButton" label="OK"/>
        <mx:Button id="cancelButton" label="Cancel"/>
    </mx:HBox>
</mx:TitleWindow>
```

The main application defines the event listeners and registers them with the controls defined within the pop-up window:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainArrayEntryFormEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import flash.events.Event;
            import myComponents.ArrayEntryFormEvents;

            public var pop1:ArrayEntryFormEvents;

            public function displayForm():void {
                // Array with data for the custom control ComboBox control.
                var doctypes:Array = ["*.as", "*.mxml", "*.swc"]

                // Create the pop-up and cast the return value
                // of the createPopUp()
                // method to the ArrayEntryFormEvents custom component.
                pop1 = ArrayEntryFormEvents(
              PopUpManager.createPopUp(this, ArrayEntryFormEvents, true));
                // Set TitleWindow properties.
                pop1.title="Select File Type";
                pop1.showCloseButton=true;
                // Set the event listeners for
                // the ArrayEntryFormEvents component.
                pop1.addEventListener("close", removeMe);
                pop1["cancelButton"].addEventListener("click", removeMe);
                pop1["okButton"].addEventListener("click", submitData);

                // Set properties of the ArrayEntryFormEvents custom control.
                pop1.myArray = doctypes;
                PopUpManager.centerPopUp(pop1);
            }

            // OK button click event listener.
            // Sets the target component in the application to the
            // selected ComboBox item value.
            private function submitData(event:Event):void {
                ti1.text = String(pop1.cb1.selectedItem);
                removeMe(event);
            }
            // Cancel button click event listener.
            private function removeMe(event:Event):void {
                PopUpManager.removePopUp(pop1);
            }
        ]]>
    </fx:Script>
    <mx:VBox>
        <mx:TextInput id="ti1" text=""/>
    </mx:VBox>
    <mx:Button id="b1" label="Select File Type" click="displayForm();"/>
</s:Application>
```

Using the addPopUp() method

You can use the `addPopUp()` method of the PopUpManager to create a pop-up window without defining a custom component. This method takes an instance of any class that implements IFlexDisplayObject. Because it takes a class instance, not a class, you can use ActionScript code in an `<fx:Script>` block to create the component instance to pop up, rather than as a separate custom component.

Using the `addPopUp()` method may be preferable to using the `createPopUp()` method if you have to pop up a simple dialog box that is never reused elsewhere. However, it is not the best coding practice if the pop-up is complex or cannot be reused elsewhere.

The following example creates a pop-up with `addPopUp()` method and adds a Button control to that window that closes the window when you click it:

```
<?xml version="1.0"?>
<!-- containers\layouts\MyPopUpButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600" width="600" >
    <fx:Script>
        <![CDATA[
            import mx.containers.TitleWindow;
            import flash.events.*;
            import mx.managers.PopUpManager;
            import mx.controls.Button;
            import mx.core.IFlexDisplayObject;

            // The variable for the TitleWindow container
            public var myTitleWindow:TitleWindow = new TitleWindow();
            // Method to instantiate and display a TitleWindow container.
            // This is the initial Button control's click event handler.
            public function openWindow(event:MouseEvent):void {
                // Set the TitleWindow container properties.
                myTitleWindow = new TitleWindow();
                myTitleWindow.title = "My Window Title";
                myTitleWindow.width= 220;
                myTitleWindow.height= 150;
                // Call the method to add the Button control to the
                // TitleWindow container.
                populateWindow();
                // Use the PopUpManager to display the TitleWindow container.
                PopUpManager.addPopUp(myTitleWindow, this, true);
            }

            // The method to create and add the Button child control to the
            // TitleWindow container.
            public function populateWindow():void {
                var btn1:Button = new Button();
                btn1.label="close";
                btn1.addEventListener(MouseEvent.CLICK, closeTitleWindow);
                myTitleWindow.addChild(btn1);
            }

            // The method to close the TitleWindow container.
            public function closeTitleWindow(event:MouseEvent):void {
                PopUpManager.removePopUp(event.currentTarget.parent);
            }
        ]]>
    </fx:Script>
    <mx:Button label="Open Window" click="openWindow(event);"/>
</s:Application>
```

You can make any component pop up. The following example pops up a TextArea control. The example resizes the control, and listens for a Shift-click to determine when to close the TextArea. Whatever text you type in the TextArea is stored in a Label control in the application when the pop-up window is removed.

```
<?xml version="1.0"?>
<!-- containers\layouts\MyPopUpTextArea.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import mx.controls.TextArea;
            import mx.core.IFlexDisplayObject;

            public var myPopUp:TextArea

            public function openWindow(event:MouseEvent):void {
                myPopUp = new TextArea();
                myPopUp.width= 220;
                myPopUp.height= 150;
                myPopUp.text = "Hold down the Shift key, and " +
                    "click in the TextArea to close it.";
                myPopUp.addEventListener(MouseEvent.CLICK, closeWindow);
                PopUpManager.addPopUp(myPopUp, this, true);
                PopUpManager.centerPopUp(myPopUp);
            }

            public function closeWindow(event:MouseEvent):void {
                if (event.shiftKey) {
                    label1.text = myPopUp.text;
        PopUpManager.removePopUp(IFlexDisplayObject(event.currentTarget));
                }
            }
        ]]>
    </fx:Script>
    <mx:VBox>
        <mx:Button id="b1" label="Create TextArea Popup"
            click="openWindow(event);"/>
        <mx:Label id="label1"/>
    </mx:VBox>
</s:Application>
```

MX navigator containers

MX navigator containers control user movement, or navigation, among multiple children where the children are other MX containers. The individual child containers of the navigator container oversee the layout and positioning of their children; the navigator container does not oversee layout and positioning.

About MX navigator containers

An MX navigator container controls user movement through a group of child MX containers. For example, a TabNavigator container lets you select the visible child container by using a set of tabs.

***Note:** The direct children of an MX navigator container must be MX containers, either MX layout or MX navigator containers, or a Spark NavigatorContent container. You cannot directly nest a control or a Spark container other than the Spark NavigatorContent container in an MX navigator.*

Adobe® Flex® provides the following navigator containers:

- ViewStack
- TabNavigator
- Accordion

Using Spark containers in an MX navigator container

The children of a navigator container must be MX containers or a Spark NavigatorContent container. The following example uses the Spark NavigatorContent container as a child of an Accordion container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkNavContent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:Accordion>
        <s:NavigatorContent label="Pane 1"
            width="100" height="100">
            <s:layout>
                <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
            </s:layout>
            <s:Label text="Text for Pane 1"/>
            <s:Button label="Button 1"/>
        </s:NavigatorContent>
        <s:NavigatorContent label="Pane 2"
            width="100" height="100">
            <s:layout>
                <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
            </s:layout>
            <s:Label text="Text for Pane 2"/>
            <s:Button label="Button 2"/>
        </s:NavigatorContent>
        <s:NavigatorContent label="Pane 3"
            width="100" height="100">
            <s:layout>
                <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
            </s:layout>
            <s:Label text="Text for Pane 3"/>
            <s:Button label="Button 3"/>
        </s:NavigatorContent>
    </mx:Accordion>
</s:Application>
```

For more information on the NavigatorContent container, see “The Spark NavigatorContent container” on page 433.

To use a Spark container other than the Spark NavigatorContent container as the child of a navigator, wrap it in an MX container or in the Spark NavigatorContent container. The following wraps a Spark SkinnableContainer container in an MX Box container:

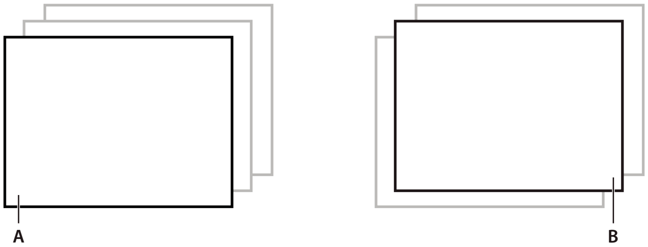
```
<?xml version="1.0"?>
<!-- containers\navigators\AccordionSimpleSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Accordion id="accordion1" height="250">
        <mx:Box label="Accordion Panel 1">
            <s:SkinnableContainer>
                <s:TextInput text="Text 1"/>
            </s:SkinnableContainer>
        </mx:Box>
        <mx:Box label="Accordion Panel 2">
            <s:SkinnableContainer>
                <s:Button label="Button 2"/>
            </s:SkinnableContainer>
        </mx:Box>
        <mx:Box label="Accordion Panel 3">
            <s:SkinnableContainer>
                <s:CheckBox label="CheckBox 3"/>
            </s:SkinnableContainer>
        </mx:Box>
    </mx:Accordion>
</s:Application>
```

This scenario lets you take advantage of the new features of the Spark containers, such as skinning and styling, which still being able to use them in MX navigators.

MX ViewStack navigator container

A ViewStack navigator container is made up of a collection of child containers that are stacked on top of each other, with only one container visible, or active, at a time. The ViewStack container does not define a built-in mechanism for users to switch the currently active container; you must use a LinkBar, TabBar, ButtonBar, or ToggleButtonBar control or build the logic yourself in ActionScript to let users change the currently active child. For example, you can define a set of Button controls that switch among the child containers.

The following image shows the stacked child containers in a ViewStack container:



A. *Child container 1 active* ***B.*** *Child container 0 active*

On the left, you see a ViewStack container with the first child active. The index of a child in a ViewStack container is numbered from 0 to *n* - 1, where *n* is the number of child containers. The container on the right is a ViewStack container with the second child container active.

For complete reference information, see ViewStack in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a ViewStack container

You use the <mx:ViewStack> tag to define a ViewStack container, and you define the child containers in the tag body. You use the following properties of the ViewStack container to control the active child container:

`selectedIndex` The index of the currently active container if one or more child containers are defined. The value of this property is -1 if no child containers are defined. The index is numbered from 0 to `numChildren` - 1, where `numChildren` is the number of child containers in the ViewStack container. Set this property to the index of the container that you want active.

You can use the `selectedIndex` property of the `<mx:ViewStack>` tag to set the default active container when your application starts. The following example sets the index of the default active container to 1:

```
<mx:ViewStack id="myViewStack" selectedIndex="1">
```

The following example uses ActionScript to set the `selectedIndex` property so that the active child container is the second container in the stack:

```
myViewStack.selectedIndex=1;
```

`selectedChild` The currently active container if one or more child containers are defined. The value of this property is `null` if no child containers are defined. Set this property in ActionScript to the identifier of the container that you want active.

You can set this property only in an ActionScript statement, not in MXML.

The following example uses ActionScript to set the `selectedChild` property so that the active child container is the child container with an identifier of `search`:

```
myViewStack.selectedChild=search;
```

`numChildren` Contains the number of child containers in the ViewStack container.

The following example uses the `numChildren` property in an ActionScript statement to set the active child container to the last container in the stack:

```
myViewStack.selectedIndex=myViewStack.numChildren-1;
```

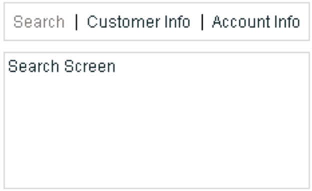
***Note:** The default creation policy for all containers, except the Application container, is the policy of the parent container. The default policy for the Application container is auto. In most cases, therefore, the View Stack control's children are not created until they are selected. You cannot set the `selectedChild` property to a child that is not yet created.*

The following example creates a ViewStack container with three child containers. The example also defines three Button controls that when clicked, select the active child container:

```
<?xml version="1.0"?>
<!-- containers\navigators\VSSimple.mxml  -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!-- Create a VGroup container so the container for
        the buttons appears above the ViewStack container. -->
    <s:VGroup>
        <!-- Create an HBox container to hold the three buttons. -->
        <mx:HBox borderStyle="solid">
            <!-- Define the three buttons.
                Each uses the child container identifier
                to set the active child container. -->
            <s:Button id="searchButton"
                label="Search Screen"
                click="myViewStack.selectedChild=search;"/>
            <s:Button id="cInfoButton"
                label="Customer Info Screen"
                click="myViewStack.selectedChild=custInfo;"/>
            <s:Button id="aInfoButton"
                label="Account Info Screen"
                click="myViewStack.selectedChild=accountInfo;"/>
        </mx:HBox>
        <!-- Define the ViewStack and the three child containers and have it
            resize up to the size of the container for the buttons. -->
        <mx:ViewStack id="myViewStack"
            borderStyle="solid" width="100%">
            <mx:Canvas id="search" label="Search">
                <mx:Label text="Search Screen"/>
            </mx:Canvas>
            <mx:Canvas id="custInfo" label="Customer Info">
                <mx:Label text="Customer Info"/>
            </mx:Canvas>
            <mx:Canvas id="accountInfo" label="Account Info">
                <mx:Label text="Account Info"/>
            </mx:Canvas>
        </mx:ViewStack>
    </s:VGroup>
</s:Application>
```

When this example loads, the three Button controls appear, and the first child container defined in the ViewStack container is active. Select a Button control to change the active container.

You can also use an MX LinkBar, TabBar, ButtonBar, or ToggleButtonBar control or the Spark ButtonBar or TabBar control to set the active child of a ViewStack container. These controls can determine the number of child containers in a ViewStack container, and then create a horizontal or vertical set of links, tabs, or buttons that let the user select the active child, as the following image shows:



The items in the LinkBar control correspond to the values of the `label` property of each child of the ViewStack container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\navigators\VSLinkBar.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:VGroup>
        <!-- Create a LinkBar control to navigate
            the ViewStack container. -->
        <mx:LinkBar dataProvider="{myViewStack}" borderStyle="solid"/>
        <!-- Define the ViewStack and the three child containers. -->
        <mx:ViewStack id="myViewStack"
            borderStyle="solid"
            width="100%">

            <mx:Canvas id="search" label="Search">
                <mx:Label text="Search Screen"/>
            </mx:Canvas>
            <mx:Canvas id="custInfo" label="Customer Info">
                <mx:Label text="Customer Info"/>
            </mx:Canvas>
            <mx:Canvas id="accountInfo" label="Account Info">
                <mx:Label text="Account Info"/>
            </mx:Canvas>
        </mx:ViewStack>
    </s:VGroup>
</s:Application>
```

You provide only a single property to the navigator bar control, `dataProvider`, to specify the name of the ViewStack container associated with it. For more information on the LinkBar control, see “LinkBar control” on page 693. For more information on the TabBar control, see “MX TabBar control” on page 720. For more information on the ButtonBar and ToggleButtonBar controls, see “MX ButtonBar and MX ToggleButtonBar controls” on page 656.

You can also use the Spark ButtonBar control to set the active child of a ViewStack container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\navigators\VSSparkButtonBar.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:VGroup>
        <!-- Create a Spark ButtonBar control to navigate
            the ViewStack container. -->
        <s:ButtonBar dataProvider="{myViewStack}"/>
        <!-- Define the ViewStack and the three child containers. -->
        <mx:ViewStack id="myViewStack"
            borderStyle="solid"
            width="100%">

            <s:NavigatorContent id="search" label="Search">
                <s:Label text="Search Screen"/>
            </s:NavigatorContent>
            <s:NavigatorContent id="custInfo" label="Customer Info">
                <s:Label text="Customer Info"/>
            </s:NavigatorContent>
            <s:NavigatorContent id="accountInfo" label="Account Info">
                <s:Label text="Account Info"/>
            </s:NavigatorContent>
        </mx:ViewStack>
    </s:VGroup>
</s:Application>
```

For more information on the Spark ButtonBar control, see the “Spark ButtonBar and TabBar controls” on page 529.

Sizing the children of a ViewStack container

The default width and height of a ViewStack container is the width and height of the first child. A ViewStack container does not change size every time you change the active child.

You can use the following techniques to control the size of a ViewStack container so that it displays all the components inside its children:

- Set explicit `width` and `height` properties for all children to the same fixed values.
- Set percentage-based `width` and `height` properties for all children to the same fixed values.
- Set `width` and `height` properties for the ViewStack container to a fixed or percentage-based value.

  The technique that you use is based on your application and the content of your ViewStack container.

Applying effects to a ViewStack container

You can assign effects to the ViewStack container or to its children. For example, if you assign the WipeRight effect to the `creationCompleteEffect` property of the ViewStack control, Flex plays the effect once when the ViewStack first appears.

To specify the effects that run when the ViewStack changes children, use the children’s `hideEffect` and `showEffect` properties. The effect specified by the `hideEffect` property plays as a container is being hidden, and the effect specified by the `showEffect` property plays as the newly visible child appears. The ViewStack container waits for the completion of the effect specified by the `hideEffect` property for the container that is being hidden before it reveals the new child container.

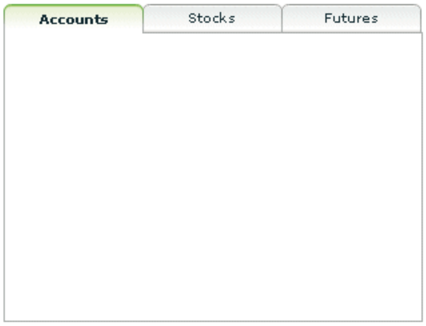
The following example runs a WipeRight effect when the ViewStack container is first created, and runs a WipeDown effect when each child is hidden and a WipeUp effect when a new child appears.

```
<?xml version="1.0"?>
<!-- containers\navigators\VSLinkEffects.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <mx:WipeUp id="myWU" duration="300"/>
        <mx:WipeDown id="myWD" duration="300"/>
        <mx:WipeRight id="myWR" duration="300"/>
    </fx:Declarations>
    <s:VGroup>
        <mx:LinkBar dataProvider="{myViewStack}"
            borderStyle="solid"
            backgroundColor="#EEEEFF"/>
        <mx:ViewStack id="myViewStack"
            borderStyle="solid"
            width="100%"
            creationCompleteEffect="{myWR}">
            <mx:Canvas id="search"
                label="Search"
                hideEffect="{myWD}"
                showEffect="{myWU}">
                <mx:Label text="Search Screen"/>
            </mx:Canvas>
            <mx:Canvas id="custInfo"
                label="Customer Info"
                hideEffect="{myWD}"
                showEffect="{myWU}">
                <mx:Label text="Customer Info"/>
            </mx:Canvas>
            <mx:Canvas id="accountInfo"
                label="Account Info"
                hideEffect="{myWD}"
                showEffect="{myWU}">
                <mx:Label text="Account Info"/>
            </mx:Canvas>
        </mx:ViewStack>
    </s:VGroup>
</s:Application>
```

Notice that the `showEffect` property of a container is only triggered when the container's visibility changes from `false` to `true`. Therefore, you use the `creationCompleteEffect` property to trigger the effect when Flex first creates the component.

MX TabNavigator container

A TabNavigator container creates and manages a set of tabs, which you use to navigate among its children. The children of a TabNavigator container are other containers. The TabNavigator container creates one tab for each child. When the user selects a tab, the TabNavigator container displays the associated child, as the following image shows:



The TabNavigator container is a child class of the ViewStack container and inherits much of its functionality. For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a TabNavigator container

You use the <mx:TabNavigator tag to define a TabNavigator container. Only one child of the TabNavigator container is visible at a time. Users can make any child the selected child by selecting its associated tab or by using keyboard navigation controls. Whenever the user changes the current child, the TabNavigator container broadcasts a `change` event.

The TabNavigator container automatically creates a tab for each of its children and determines the tab text from the `label` property of the child, and the tab icon from the `icon` property of the child. The tabs are arranged left to right in the order determined by the child indexes. All tabs are visible, unless they do not fit within the width of the TabNavigator container.

If you disable a child of a TabNavigator container by setting its `enabled` property to `false`, you also disable the associated tab.

The following code creates the TabNavigator container in the image in “MX TabNavigator container” on page 636:

```
<?xml version="1.0"?>
<!-- containers\navigators\TNSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:TabNavigator borderStyle="solid" >
        <mx:VBox label="Accounts"
            width="300"
            height="150">
            <!-- Accounts view goes here. -->
        </mx:VBox>
        <mx:VBox label="Stocks"
            width="300"
            height="150">
            <!-- Stocks view goes here. -->
        </mx:VBox>
        <mx:VBox label="Futures"
            width="300"
            height="150">
            <!-- Futures view goes here. -->
        </mx:VBox>
    </mx:TabNavigator>
</s:Application>
```

You can also set the currently active child by using the `selectedChild` and `selectedIndex` properties inherited from the ViewStack container as follows:

- `numChildren` The index of the currently active container if one or more child containers are defined. The index is numbered from 0 to `numChildren` - 1, where `numChildren` is the number of child containers in the TabNavigator container. Set this property to the index of the container that you want active.
- `selectedChild` The currently active container, if one or more child containers are defined. This property is `null` if no child containers are defined. Set this property to the identifier of the container that you want active. You can set this property only in an ActionScript statement, not in MXML.

For more information on the `selectedChild` and `selectedIndex` properties, including examples, see “MX ViewStack navigator container” on page 630.

You use the `showEffect` and `hideEffect` properties of the children of a TabNavigator container to specify an effect to play when the user changes the currently active child. The following example plays the WipeRight effect each time the selected tab changes:

```
<?xml version="1.0"?>
<!-- containers\navigators\TNEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <mx:WipeRight id="myWR"/>
    </fx:Declarations>
    <mx:TabNavigator>
        <mx:VBox label="Accounts"
            width="300"
            height="150"
            showEffect="{myWR}">
            <!-- Accounts view goes here. -->
            <mx:Text text="This is a text control."/>
        </mx:VBox>
        <mx:VBox label="Stocks"
            width="300"
            height="150"
            showEffect="{myWR}">
            <!-- Stocks view goes here. -->
            <mx:Text text="This is a text control."/>
        </mx:VBox>
        <mx:VBox label="Futures"
            width="300"
            height="150"
            showEffect="{myWR}">
            <!-- Futures view goes here. -->
            <mx:Text text="This is a text control."/>
        </mx:VBox>
    </mx:TabNavigator>
</s:Application>
```

Sizing the children of a TabNavigator container

The default width and height of a TabNavigator container is the width and height of the first child. A TabNavigator container does not change size every time you change the active child.

You can use the following techniques to control the size of a TabNavigator container so that it displays all the components inside its children:

- Set explicit `width` and `height` properties for all children to the same fixed values.
- Set percentage-based `width` and `height` properties for all children to the same fixed values.
- Set explicit or percentage-based `width` and `height` properties for the TabNavigator container.

  The method that you use is based on your application and the content of your TabNavigator container.

TabNavigator container Keyboard navigation

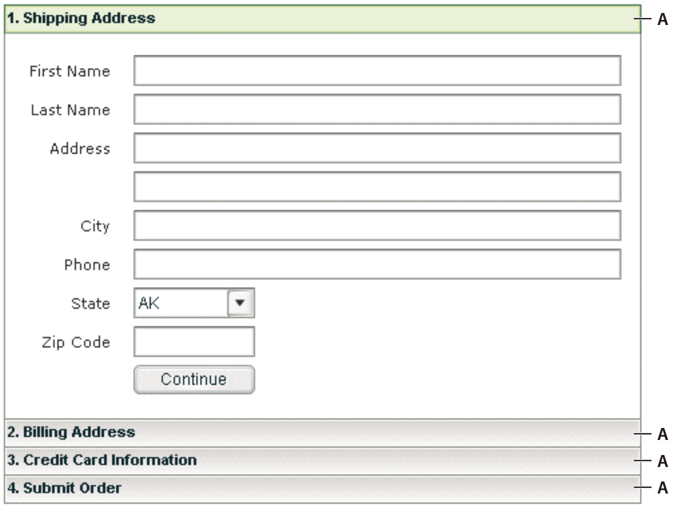
When a TabNavigator container has focus, Flex processes keystrokes as the following table describes:

Key	Action
Down Arrow Right Arrow	Gives focus to the next tab, wrapping from last to first, without changing the selected child.
Up Arrow Left Arrow	Gives focus to the previous tab, wrapping from first to last, without changing the selected child.
Page Down	Selects the next child, wrapping from last to first.
Page Up	Selects the previous child, wrapping from first to last.
Home	Selects the first child.
End	Selects the last child.
Enter Spacebar	Selects the child associated with the tab displaying focus.

MX Accordion navigator container

Forms are a basic component of many applications. However, users have difficulty navigating through complex forms, or moving back and forth through multipage forms. Sometimes, forms can be so large that they do not fit onto a single screen.

Flex includes the Accordion navigator container, which can greatly improve the look and navigation of a form. The Accordion container defines a sequence of child panels, but displays only one panel at a time. The following image shows an example of an Accordion container:



A. *Accordion container navigation button* *A.* *Accordion container navigation button*

To navigate a container, the user clicks the navigation button that corresponds to the child panel that they want to access. Accordion containers let users access the child panels in any order to move back and forth through the form. For example, when the user is in the Credit Card Information panel, they might decide to change the information in the Billing Address panel. To do so, they navigate to the Billing Address panel, edit the information, and then navigate back to the Credit Card Information panel.

In HTML, a form that contains shipping address, billing address, and credit card information is often implemented as three separate pages, which requires the user to submit each page to the server before moving on to the next. An Accordion container can organize the information on three child panels with a single Submit button. This architecture minimizes server traffic and lets the user maintain a better sense of progress and context.

***Note:** An empty Accordion container with no child panels cannot take focus.*

Although Accordion containers are useful for working with forms and Form containers, you can use any Flex component within a child panel of an Accordion container. For example, you could create a catalog of products in an Accordion container, where each panel contains a group of similar products.

For complete reference information, see Accordion in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating an Accordion container

You use the `<mx:Accordion>` tag to define an Accordion container. In the Accordion container, you define one container for each child panel. For example, if the Accordion container has four child panels that the correspond to four parts of a form, you define each child panel by using the Form container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\navigators\AccordionSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Accordion id="accordion1" height="250">
        <mx:Form id="shippingAddress" label="1. Shipping Address">
            <mx:FormItem id="sfirstNameItem" label="First Name">
                <mx:TextInput id="sfirstName"/>
            </mx:FormItem>
            <!-- Additional contents goes here. -->
        </mx:Form>
        <mx:Form id="billingAddress" label="2. Billing Address">
            <!-- Form contents goes here. -->
        </mx:Form>
        <mx:Form id="creditCardInfo" label="3. Credit Card Information">
            <!-- Form contents goes here. -->
        </mx:Form>
        <mx:Form id="submitOrder"   label="4. Submit Order">
            <!-- Form contents goes here. -->
        </mx:Form>
    </mx:Accordion>
</s:Application>
```

This example defines each child panel by using a Form container. However, you can use any container to define a child panel.

***Note:** You can use any container to define child panels. Some containers, such as an Accordion container, do not belong in child panels.*

Accordion container Keyboard navigation

When an Accordion container has focus, Flex processes keystrokes as the following table describes:

Key	Action
Down Arrow Right Arrow	Gives focus to the next button, wrapping from last to first, without changing the selected child.
Up Arrow Left Arrow	Gives focus to the previous button, wrapping from first to last, without changing the selected child.
Page Up	Moves to the previous child panel, wrapping from first to last.
Page Down	Moves to the next child panel, wrapping from last to first.
Home	Moves to the first child panel.
End	Moves to the last child panel.
Enter Spacebar	Selects the child associated with the tab displaying focus.

Using Button controls to navigate an Accordion container

The simplest way for users to navigate the panels of an Accordion container is to click the navigator button for the desired panel. However, you can create additional navigation Button controls, such as Back and Next, to make it easier for users to navigate.

Navigation Button controls use the following properties of the Accordion container to move among the child panels:

Property	Description
`numChildren`	Contains the total number of child panels defined in an Accordion container.
`selectedIndex`	The index of the currently active child panel. Child panels are numbered from 0 to `numChildren` - 1. Setting the `selectedIndex` property changes the currently active panel.
`selectedChild`	The currently active child container if one or more child containers are defined. This property is `null` if no child containers are defined. Set this property to the identifier of the container that you want active. You can set this property only in an ActionScript statement, not in MXML.

For more information on these properties, see “MX ViewStack navigator container” on page 630.

You can use the following two Button controls, for example, in the second panel of an Accordion container, panel number 1, to move back to panel number 0 or ahead to panel number 2:

```
<?xml version="1.0"?>
<!-- containers\navigators\AccordionButtonNav.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Accordion id="accordion1" height="250">
        <mx:Form id="shippingAddress" label="1. Shipping Address">
            <mx:FormItem id="sfirstNameItem" label="First Name">
                <s:TextInput id="sfirstName"/>
            </mx:FormItem>
        </mx:Form>
        <mx:Form id="billingAddress" label="2. Billing Address">
            <s:Button id="backButton"
                label="Back"
                click="accordion1.selectedIndex=0;"/>
            <s:Button id="nextButton"
                label="Next"
                click="accordion1.selectedIndex=2;"/>
        </mx:Form>
        <mx:Form id="creditCardInfo" label="3. Credit Card Information">
        </mx:Form>
    </mx:Accordion>
</s:Application>
```

You can also use relative location with navigation buttons. The following Button controls move forward and backward through Accordion container panels based on the current panel number:

```
<s:Button id="backButton" label="Back"
    click="accordion1.selectedIndex = accordion1.selectedIndex - 1;"/>
<s:Button id="nextButton" label="Next"
    click="accordion1.selectedIndex = accordion1.selectedIndex + 1;"/>
```

For the Next Button control, you also can use the `selectedChild` property to move to the next panel based on the value of the `id` property of the panel's container, as the following code shows:

```
<s:Button id="nextButton" label="Next"
    click="accordion1.selectedChild=creditCardInfo;"/>
```

The following Button control opens the last panel in the Accordion container:

```
<s:Button id="lastButton" label="Last"
    click="accordion1.selectedIndex = accordion1.numChildren - 1;"/>
```

Handling child button events

The Accordion container can recognize an event when the user changes the current panel. The Accordion container broadcasts a `change` event when the user changes the child panel, either by clicking a button or pressing a key, such as the Page Down key.

***Note:** A `change` event is not dispatched when the child panel changes programmatically; for example, the `change` event is not dispatched when you use the buttons in change panels (see "Using Button controls to navigate an Accordion container" on page 641). However, the `valueCommit` event is dispatched.*

You can register an event handler for the `change` event by using the `change` property of the `<mx:Accordion>` tag, or by registering the handler in ActionScript. The following example logs the change event to flashlog.txt each time the user changes panels:

```
<mx:Accordion id="accordion1" height="450" change="trace('change');">
```

Controlling the appearance of accordion buttons

The buttons on an Accordion container are rendered by the AccordionHeader class, which is a subclass of Button, and has the same style properties as the Button class.

To change the style of an Accordion button, call the Accordion class `getHeaderAt()` method to get a reference to a child container's button, and then call the button's `setStyle()` method to set the style. The following example uses this technique to set a different text color for each of the Accordion buttons:

```
<?xml version="1.0"?>
<!-- containers\navigators\AccordionStyling.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600"
    height="600"
    creationComplete="setButtonStyles();">
    <fx:Script>
        <![CDATA[
            public function setButtonStyles():void {
                comp.getHeaderAt(0).setStyle('color', 0xAA0000);
                comp.getHeaderAt(1).setStyle('color', 0x00AA00);
            }
        ]]>
    </fx:Script>
    <mx:Accordion id="comp">
        <mx:VBox label="First VBox">
            <mx:TextInput/>
            <mx:Button label="Button 1"/>
        </mx:VBox>
        <mx:VBox label="Second VBox">
            <mx:TextInput/>
            <mx:Button label="Button 2"/>
        </mx:VBox>
    </mx:Accordion>
</s:Application>
```

You can also control the appearance of the buttons by using the `headerStyleName` style property of the Accordion class. For more information, see Accordion in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

UI Controls

UI Controls are user-interface components such as Button, TextArea, and CheckBox controls. Adobe® Flex® has two types of controls: basic and data provider. For information on data provider controls, see "MX data-driven controls" on page 943.

About UI controls

UI controls are user-interface components, such as Button, TextArea, and CheckBox controls. You place UI controls in containers, which are user-interface components that provide a hierarchical structure for controls and other containers. Typically, you define a container, and then insert UI controls or other containers in it.

At the root of your MXML application is the `<s:Application>` tag. This tag represents a base container that covers the entire Adobe® Flash® Player or Adobe AIR™ drawing surface. You can place controls or containers directly under the `<s:Application>` tag or in other containers. For more information on containers, see “Introduction to containers” on page 326.

Flex provides a set of UI controls built for the Spark architecture. To take advantage of the skinning architecture in Spark, use the Spark controls whenever possible. The MX controls are also supported. You can use a combination of MX and Spark controls in an application.

To use Spark controls, include the Spark namespace (default prefix: `s`) in your Application tag. To use MX controls, include the MX namespace (default prefix: `mx`) in your Application tag.

```
xmlns:s = "library://ns.adobe.com/flex/spark"
xmlns:mx = "library://ns.adobe.com/flex/mx"
```

Most controls have the following characteristics:

- MXML API for declaring the control and the values of its properties and events
- ActionScript API for calling the control’s methods and setting its properties at run time
- Customizable appearance by using styles, skins, and fonts

The MXML and ActionScript APIs let you create and configure a control. The following MXML code example creates a TextInput control in a Form container:

```
<?xml version="1.0" encoding="utf-8"?>
<!--controls\TextInputInForm.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Form width="400" height="100">
        <s:FormItem label="Card Name">
            <s:TextInput id="cardName"/>
        </s:FormItem>
    </s:Form>

</s:Application>
```

Although you commonly use MXML as the language for building applications in Flex, you can also use ActionScript to configure controls. For example, the following code example populates a Spark DataGrid control by providing an Array of items as the value of the DataGrid control’s `dataProvider` property:

```
<?xml version="1.0" encoding="utf-8"?>
<!--controls\DataGridConfigAS.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            private function myGrid_initialize():void {
              myGrid.dataProvider = new ArrayCollection([
                {Artist:'Steve Goodman', Album:'High and Outside', Price:8.99},
                {Artist:'Carole King', Album:'Tapestry', Price:11.99},
                {Artist:'The Beach Boys', Album:'Pet Sounds', Price:13.99},
                {Artist:'Original Cast', Album:'Camelot', Price:9.99} ]);
            }
        ]]>
    </fx:Script>

    <s:DataGrid id="myGrid"
        width="350" height="150"
        color="#7B0974"
        creationComplete="myGrid_initialize();"/>

</s:Application>
```

Text controls

Several Flex components display text or take text input, as the following table shows:

Type of text	Spark control	MX control
Editable, single-line text	TextInput	TextInput
Editable, multiline text	TextArea	TextArea
Noneditable, single-line text	Label	Label
Noneditable, multiline text	Label	Text
Noneditable, richly formatted text	RichText	n/a
Editable, richly formatted text	RichEditableText	n/a
Compound control that contains a multiline text field and controls that let you format text by selecting such characteristics as font, size, weight, and alignment	n/a	RichTextEditor

These controls can display plain text that all has the same appearance. The controls can also display rich text formatted by using a subset of the standard HTML formatting tags. For information on using text controls, see “MX text controls” on page 805.

Data provider controls

Several Flex components, such as the MX DataGrid and Tree controls, and the Spark ComboBox and List controls, take input data from a data provider. A *data provider* is a collection of objects, similar to an array. For example, a Tree control reads data from the data provider to define the structure of the tree and any associated data assigned to each tree node.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same data provider, switch data providers for a component at runtime, and modify the data provider so that changes are reflected by all components that use the data provider.

Consider that the data provider is the model, and the components are the view onto the model. By separating the model from the view, you can change one without changing the other.

Menu controls

Several MX controls create or interact with menus, as the following table shows:

MX control (in mx.controls package)	Description
Menu	A visual menu that can have cascading submenus
MenuBar	A horizontal bar with multiple submenus
PopUpMenuButton	A Menu control that opens when you click a button

In Spark, you can create custom menu-type controls using the “PopUpAnchor control” on page 698.

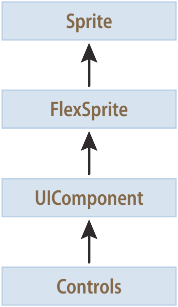
For information on menu controls, see “Menu-based controls” on page 985.

Working with controls

Controls share a common class hierarchy. Therefore, you use a similar procedure to configure all controls.

Class hierarchy of controls

Controls are ActionScript objects derived from the flash.display.Sprite, mx.core.FlexSprite, and mx.core.UIComponent classes, as the following diagram shows. Controls inherit the properties, methods, events, styles, skin parts, skin states, and effects of these superclasses:



Class hierarchy of controls

The Sprite, FlexSprite, and UIComponent classes are the base classes for all Flex components. Subclasses of the UIComponent class can have shape, draw themselves, and be invisible. Each subclass can participate in tabbing. Subclasses can accept low-level events like keyboard and mouse input. They can be disabled so that they do not receive mouse and keyboard input.

For information on the interfaces inherited by controls from the Sprite and UIComponent classes, see “Visual components” on page 280.

Sizing controls

All controls define rules for determining their size in an application. For example, a Button control sizes itself to fit its label text and optional icon image. An Image control sizes itself to the size of the imported image. Each control has a default height and a default width. The default size of each standard control is specified in the description of each control.

The default size of a control is not necessarily a fixed value. For example, for a Button control, the default size is large enough to fit its label text and optional icon image. At runtime, Flex calculates the default size of each control and, by default, does not resize a control from its default size.

Set the `height` and `width` attributes in MXML to percentages, such as 50%, or the `percentHeight` and `percentWidth` properties in ActionScript to percentage values, such as 50, to allow Flex to resize the control in the corresponding direction. Flex attempts to fit the control to the percentage of its parent container that you specify. If there isn't enough space available, the percentages are scaled, while retaining their relative values.

For example, you can set the width of a comments box to scale with its parent container as the parent container changes size:

```
<s:TextArea id="comments" width="100%" height ="20"/>
```

You can also specify explicit sizes for a control in MXML or ActionScript by setting the `height` and `width` properties to numeric pixel values. The following example sets the height and width of the `addr2` TextInput control to 20 pixels and 100 pixels, respectively:

```
<s:TextInput id="addr2" width="100" height ="20"/>
```

To resize a control at runtime, use ActionScript to set its `width` and `height` properties. For example, the click event listener for the following Button control increases the value of the `width` property of the `addr2` TextInput control by ten pixels:

```
<s:Button id="button1" label="Slide" height="20"
    click="addr2.width+=10;"/>
```

***Note:** The preceding technique works even if the `width` property was originally set as a percentage value. The stored values of the `width` and `height` properties are always in pixels.*

Many components have arbitrarily large maximum sizes, which means that Flex can make them as large as necessary to fit the requirements of your application. While some components have a defined nonzero minimum size, most have a minimum size of 0. You can use the `maxHeight`, `maxWidth`, `minHeight`, and `minWidth` properties to set explicit size ranges for each component.

For more information on sizing components, see "Laying out components" on page 359.

Positioning controls

You place controls inside containers. Most containers have predefined layout rules that automatically determine the position of their children. The Spark Group container positions children using BasicLayout, which is absolute positioning. The HGroup container positions children with a horizontal layout. The VGroup container positions children with a vertical layout. The MX Canvas container absolutely positions its children.

To absolutely position a control, you set its `x` and `y` properties to specific horizontal and vertical pixel coordinates within the container. These coordinates are relative to the upper-left corner of the container, where the upper-left corner is at coordinates (0,0). Values for `x` and `y` can be positive or negative integers. You can use negative values to place a control outside the visible area of the container. You can then use ActionScript to move the child to the visible area, possibly as a response to an event.

The following example places the TextInput control 150 pixels to the right and 150 pixels down from the upper-left corner of a Canvas container:

```
<s:TextInput id="addr2" width="100" height ="20" x="150" y="150"/>
```

To reposition a control within an absolutely-positioned container at runtime, you set its `x` and `y` properties. For example, the `click` event listener for the following Button control moves the TextInput control down ten pixels from its current position:

```
<s:Button id="button1" label="Slide" height="20" x="0" y="250"
    click="addr2.y = addr2.y+10;"/>
```

For detailed information about control positioning, including container-relative positioning, see "Laying out components" on page 359.

Changing the appearance of controls

Styles, skins, and fonts let you customize the appearance of controls. They describe aspects of components that you want components to have in common. Each control defines a set of styles, skins, and fonts that you can set. Some of these characteristics are specific to a particular type of control, and others are more general. In Spark, the skin controls all visual elements of a component, including layout. See "About Spark skins" on page 1602.

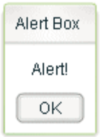
Flex provides several different ways for you to configure the appearance of your controls. For example, you can set styles for a specific control in the control's MXML tag, in CSS, or by using ActionScript. You can set styles globally for all instances of a specific control in an application by using the `<fx:Style>` tag.

A *theme* defines the appearance of an application. A theme can define something as simple as the color scheme or common font for an application. It can also be a complete reskinning of all the components. The current theme for your application defines the styles that you can set on the controls within it. That means some style properties might not always be settable. For more information, see "Styles and themes" on page 1492.

Alert control

The Alert control is part of the MX component set. There is no Spark equivalent.

All Flex components can call the static `show()` method of the Alert class to open a modal dialog box that contains a message and an optional title, buttons, and icons. The following example shows an Alert control pop-up dialog box:



The Alert control closes when you select a button in the control, or press the Escape key.

The `Alert.show()` method has the following syntax:

```
public static show(
    text:String,
    title:String=null,
    flags:uint=mx.controls.Alert.OK,
    parent:Sprite=null,
    clickListener:Function=null,
    iconClass:Class=null,
    defaultButton:uint=mx.controls.Alert.OK
):Alert
```

This method returns an Alert control object.

The following table describes the arguments of the `show()` method:

Argument	Description
`text`	(Required) Specifies the text message displayed in the dialog box.
`title`	Specifies the dialog box title. If omitted, displays a blank title bar.
`flags`	Specifies the buttons to display in the dialog box. The options are as follows: `mx.controls.Alert.OK` — OK button `mx.controls.Alert.YES` — Yes button `mx.controls.Alert.NO` — No button `mx.controls.Alert.CANCEL` — Cancel button Each option is a bit value and can be combined with other options by using the pipe '|' operator. The buttons appear in the order listed here regardless of the order specified in your code. The default value is `mx.controls.Alert.OK`.
`parent`	The parent object of the Alert control.
`clickListener`	Specifies the listener for click events from the buttons. The event object passed to this handler is an instance of the CloseEvent class. The event object contains the `detail` field, which is set to the button flag that was clicked (`mx.controls.Alert.OK`, `mx.controls.Alert.CANCEL`, `mx.controls.Alert.YES`, or `mx.controls.Alert.NO`).
`iconClass`	Specifies an icon to display to the left of the message text in the dialog box.
`defaultButton`	Specifies the default button by using one of the valid values for the `flags` argument. This is the button that is selected when the user presses the Enter key. The default value is `Alert.OK`. Pressing the Escape key triggers the Cancel or No button.

To use the Alert control, you first import the Alert class into your application, then call the `show()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:TextInput id="myInput"
                     width="150"
                     text=""/>
        <s:Button id="myButton"
                  label="Copy Text"
                  click="myText.text = myInput.text;
            Alert.show('Text Copied!', 'Alert Box', mx.controls.Alert.OK);"/>
        <s:TextInput id="myText"/>
    </s:VGroup>
</s:Application>
```

In this example, selecting the Button control copies text from the TextInput control to the TextArea control, and displays the Alert control.

You can also define an event listener for the Button control, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertSimpleEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function alertListener():void {
                myText.text = myInput.text;
                Alert.show("Text Copied!", "Alert Box", Alert.OK);
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:TextInput id="myInput"
                     width="150"
                     text=""/>
        <s:Button id="myButton"
                  label="Copy Text"
                  click="alertListener();"/>
        <s:TextInput id="myText"/>
    </s:VGroup>
</s:Application>
```

***Note:** After the `show()` method creates the dialog box, Flex continues processing of your application; it does not wait for the user to close the dialog box.*

Sizing the Alert control

The Alert control automatically sizes itself to fit its text, buttons, and icon. You can explicitly size an Alert control by using the Alert object returned from the `show()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertSize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.CloseEvent;
            // Define variable to hold the Alert object.
            public var myAlert:Alert;
            private function openAlert():void {
                myAlert = Alert.show("Copy Text?", "Alert",
                    Alert.OK    | Alert.CANCEL);
                // Set the height and width of the Alert control.
                myAlert.height=250;
                myAlert.width=250;
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:TextInput id="myInput"
                     width="150"
                     text=""/>
        <s:Button id="myButton"
                  label="Copy Text"
                  click="openAlert();"/>
        <s:TextInput id="myText"/>
    </s:VGroup>
</s:Application>
```

In this example, you set the `height` and `width` properties of the Alert object to explicitly size the control.

Using event listeners with the Alert control

The next example adds an event listener to the Alert control dialog box. An event listener lets you perform processing when the user selects a button of the Alert control. The event object passed to the event listener is of type CloseEvent.

In the next example, you only copy the text when the user selects the OK button in the Alert control:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.CloseEvent;
            private function alertListener(eventObj:CloseEvent):void {
                // Check to see if the OK button was pressed.
                if (eventObj.detail==Alert.OK) {
                    myText.text = myInput.text;
                }
            }
        ]]>
    </fx:Script>

    <s:VGroup>
        <s:TextInput id="myInput"
            width="150"
            text="" />
        <s:Button id="myButton"
            label="Copy Text"
            click='Alert.show("Copy Text?", "Alert",
            Alert.OK | Alert.CANCEL, this,
            alertListener, null, Alert.OK);'/>
        <s:TextInput id="myText"/>
    </s:VGroup>
</s:Application>
```

In this example, you define an event listener for the Alert control. Within the body of the event listener, you determine which button was pressed by examining the `detail` property of the event object. The event object is an instance of the CloseEvent class. If the user pressed the OK button, copy the text. If the user pressed any other button, or pressed the Escape key, do not copy the text.

Specifying an Alert control icon

You can include an icon in the Alert control that appears to the left of the Alert control text. This example modifies the example from the previous section to add the `Embed` metadata tag to import the icon. For more information on importing resources, see “Using ActionScript” on page 32.

```
<?xml version="1.0"?>
<!-- controls\alert\AlertIcon.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.CloseEvent;

            [Embed(source="assets/alertIcon.jpg")]
            [Bindable]
            public var iconSymbol:Class;
            private function alertListener(eventObj:CloseEvent):void {
                // Check to see if the OK button was pressed.
                if (eventObj.detail==Alert.OK) {
                    myText.text = myInput.text;
                }
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:TextInput id="myInput"
                     width="150"
                     text=""/>
        <s:Button id="myButton"
                  label="Copy Text"
                  click='Alert.show("Copy Text?", "Alert",
            Alert.OK | Alert.CANCEL, this,
                alertListener, iconSymbol,  Alert.OK );'/>
        <s:TextInput id="myText"/>
    </s:VGroup>
</s:Application>
```

Button and ToggleButton control

The Button and ToggleButton controls are part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead.

***Note:** In MX, you create a toggle button by setting the `toggle` property of the MX Button control to `true`. in Spark, the ToggleButton is a separate component.*

The Button control is a commonly used rectangular button. Button controls look like they can be pressed, and have a text label, an icon, or both on their face. You can optionally specify graphic skins for each of several Button states.

You can create a normal Button control or a ToggleButton control. A normal Button control stays in its pressed state for as long as the mouse button is down after you select it. A ToggleButton stays in the pressed state until you select it a second time. The ToggleButton control is available in Spark. In MX, the Button control contains a `toggle` property that provides similar functionality.

Buttons typically use event listeners to perform an action when the user selects the control. When a user clicks the mouse on a Button control, and the Button control is enabled, it dispatches a `click` event and a `buttonDown` event. A button always dispatches events such as the `mouseMove`, `mouseOver`, `mouseOut`, `rollOver`, `rollOut`, `mouseDown`, and `mouseUp` events whether enabled or disabled.

You can use customized graphic skins to customize your buttons to match your application’s look and functionality. You can give the Button and ToggleButton controls different skins. The control can change the image skins dynamically. The following table describes the skin states (Spark) and skin styles (MX) available for the Button and ToggleButton controls:

Spark Button skin states	Spark ToggleButton skin states	MX Button skin styles
disabled	disabled	disabledSkin
down	disabledAndSelected	downSkin
over	down	overSkin
up	downAndSelected	selectedDisabledSkin
---	over	selectedDownSkin
---	overAndSelected	selectedOverSkin
---	up	selectedUpSkin
---	upAndSelected	upSkin

Creating a Button control

You define a Button control in MXML by using the `<s:Button>` tag, as the following example shows. Specify an `id` value if you intend to refer to the button elsewhere in your MXML, either in another tag or in an ActionScript block. The following code creates a Button control with the label “Hello world!”:

```
<?xml version="1.0"?>
<!-- controls\button\ButtonLabel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Button id="button1"
        label="Hello world!"
        width="100"/>
</s:Application>
```

In Spark, all visual elements of a component, including layout, are controlled by the skin. For more information on skinning, see “About Spark skins” on page 1602.

In MX, a Button control’s icon, if specified, and label are centered within the bounds of the Button control. You can position the text label in relation to the icon by using the `labelPlacement` property, which accepts the values `right`, `left`, `bottom`, and `top`.

Sizing a Button control

By default, Flex stretches the Button control width to fit the size of its label, any icon, plus six pixels of padding around the icon. You can override this default width by explicitly setting the `width` property of the Button control to a specific value or to a percentage of its parent container. If you specify a percentage value, the button resizes between its minimum and maximum widths as the size of its parent container changes.

If you explicitly size a Button control so that it is not large enough to accommodate its label, the label is truncated and terminated by an ellipsis (...). The full label displays as a tooltip when you move the mouse over the Button control. If you have also set a tooltip by using the `toolTip` property, the tooltip is displayed rather than the label text. Text that is vertically larger than the Button control is also clipped.

If you explicitly size a Button control so that it is not large enough to accommodate its icon, icons larger than the Button control extend outside the Button control’s bounding box.

Button control user interaction

When a user clicks the mouse on a Button control, the Button control dispatches a `click` event, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\button\ButtonClick.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            protected function myBtn_clickHandler(event:MouseEvent):void {
                Alert.show("Goodbye!");
            }
        ]]>
    </fx:Script>
    <s:Button id="myBtn"
        x="83" y="92"
        label="Hello World!"
        click="myBtn_clickHandler(event)"/>

</s:Application>
```

In this example, clicking the Button triggers an Alert control to appear with a message to the user.

If a Button control is enabled, it behaves as follows:

- When the user moves the pointer over the Button control, the Button control displays its rollover appearance.
- When the user clicks the Button control, focus moves to the control and the Button control displays its pressed appearance. When the user releases the mouse button, the Button control returns to its rollover appearance.
- If the user moves the pointer off the Button control while pressing the mouse button, the control's appearance returns to the rollover state and it retains focus.
- For MX controls, if the `toggle` property is set to `true`, the state of the Button control does not change until the user releases the mouse button over the control. For the Spark ToggleButton, this statement applies to the `selected` property.

If a Button control is disabled, it displays its disabled appearance, regardless of user interaction. In the disabled state, all mouse or keyboard interaction is ignored.

Embedding an icon in a Button control

The Button controls define a style property, `icon`, that you use to add an icon to the button. A button icon can be a GIF, JPEG, PNG, SVG, or SWF file.

***Note:** For the MX Button control, icons must be embedded at compile time. You cannot load the icon at runtime.*

Use the `@Embed` syntax in the `icon` property value to embed an icon file. Or you can bind to an image that you defined within a script block by using `[Embed]` metadata. If you must reference your button graphic at runtime, you can use an Image control instead of a Button control.

For more information on embedding resources, see "Embedding assets" on page 1699.

The following code example creates a Spark Button control with a label and icon.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\button\ButtonLabelIconSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import assets.*;
            import mx.controls.Alert;
            protected function myClickHandler():void{
                Alert.show("Thanks for submitting.")
            }
        ]]>
    </fx:Script>

    <s:Button id="iconButton"
              width="100" height="30"
              x="10" y="10"
              label="Submit to"
              icon="@Embed('assets/logo.jpg')"
              click="myClickHandler();"/>
</s:Application>
```

MX ButtonBar and MX ToggleButtonBar controls

The ButtonBar control is part of both the MX and Spark component sets. Spark does not define a separate ToggleButtonBar control. You can use the Spark ButtonBar control to replicate the functionality of the MX ToggleButtonBar control. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead. For information on Spark ButtonBar, see "Spark ButtonBar and TabBar controls" on page 529.

The MX ButtonBar and ToggleButtonBar controls define a horizontal or vertical row of related buttons with a common appearance. The controls define a single event, the `itemClick` event, that is dispatched when any button in the control is selected.

The ButtonBar control defines group of buttons that do not retain a selected state. When you select a button in a ButtonBar control, the button changes its appearance to the selected state. When you release the button, it returns to the deselected state.

The ToggleButtonBar control defines a group of buttons that maintain their state, either selected or deselected. Only one button in the ToggleButtonBar control can be in the selected state. That means when you select a button in a ToggleButtonBar control, the button stays in the selected state until you select a different button.

If you set the `toggleOnClick` property of the ToggleButtonBar control to `true`, selecting the currently selected button deselects it. By default the `toggleOnClick` property is `false`.

Creating an MX ButtonBar control

You create a ButtonBar control in MXML by using the `<mx:ButtonBar>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\BBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:ButtonBar horizontalGap="5">
        <mx:dataProvider>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:dataProvider>
    </mx:ButtonBar>
</s:Application>
```

This example creates a row of four Button controls.

To create a ToggleButtonBar control, replace the `<mx:ButtonBar>` tag with the `<mx:ToggleButtonBar>` tag. For the ToggleButtonBar control, the `selectedIndex` property determines which button is selected when the control is created. The default value for `selectedIndex` is 0 and selects the leftmost button in the bar. Setting the `selectedIndex` property to -1 deselects all buttons in the bar. Otherwise, the syntax is the same for both controls.

The `dataProvider` property specifies the labels of the four buttons. You can also populate the `dataProvider` property with an Array of Objects, where each object can have up to three fields: `label`, `icon`, and `toolTip`.

In the following example, an Array of Objects specifies a label and icon for each button:

```
<?xml version="1.0"?>
<!-- controls\bar\BBarLogo.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <mx:ButtonBar horizontalGap="5">
        <mx:dataProvider>
            <fx:Object label="Flash"
                icon="@Embed(source='assets/Flashlogo.gif')"/>
            <fx:Object label="Director"
                icon="@Embed(source='assets/Dirlogo.gif')"/>
            <fx:Object label="Dreamweaver"
                icon="@Embed(source='assets/Dlogo.gif')"/>
            <fx:Object label="ColdFusion"
                icon="@Embed(source='assets/CFlogo.gif')"/>
        </mx:dataProvider>
    </mx:ButtonBar>
</s:Application>
```

A ButtonBar or ToggleButtonBar control creates Button controls based on the value of its `dataProvider` property. Even though ButtonBar and ToggleButtonBar are subclasses of Container, do not use the methods `Container.addChild()` and `Container.removeChild()` to add or remove Button controls. Instead, use methods `addItem()` and `removeItem()` to manipulate the `dataProvider` property. A ButtonBar or ToggleButtonBar control automatically adds or removes children based on changes to the `dataProvider` property.

Handling MX ButtonBar events

The MX ButtonBar and MX ToggleButtonBar controls dispatch an item`Click` event when you select a button. The event object passed to the event listener is of type ItemClickEvent. From within the event listener, you access properties of the event object to determine the index of the selected button and other information. The index of the first button is 0. For more information about the event object, see the description of the ItemClickEvent class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The ToggleButtonBar control in the following example defines an event listener, named `clickHandler()`, for the `itemClick` event.

```
<?xml version="1.0"?>
<!-- controls\bar\BBarEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.events.ItemClickEvent;
            private var savedIndex:int = 99999;

            private function clickHandler(event:ItemClickEvent):void {
                if (event.index == savedIndex) {
                    myTA.text=""
                }
                else {
                    savedIndex = event.index;
                    myTA.text="Selected button index: " +
                    String(event.index) + "\n" +
                    "Selected button label: " +
                    event.label;
                }
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <mx:ToggleButtonBar
            horizontalGap="5"
            itemClick="clickHandler(event);"
            toggleOnClick="true"
            selectedIndex="-1">

        <mx:dataProvider>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:dataProvider>
        </mx:ToggleButtonBar>
        <s:TextArea id="myTA" width="250" height="100"/>
    </s:VGroup>
</s:Application>
```

In this example, the click handler displays the index and label of the selected button in a TextArea control in response to an `itemClick` event. If you press the selected button a second time, the button is deselected, and the sample click handler clears the text area.

CheckBox control

The CheckBox control is part of both the MX and Spark component sets. While you can use the MX CheckBox control in your application, Adobe recommends that you use the Spark CheckBox control instead.

The CheckBox control is a commonly used graphical control that can contain a check mark or not. You can use CheckBox controls to gather a set of `true` or `false` values that aren't mutually exclusive.

You can add a text label to a CheckBox control and place it to the left, right, top, or bottom. Flex clips the label of a CheckBox control to fit the boundaries of the control.

When a user clicks a CheckBox control or its associated text, the CheckBox control changes its state from checked to unchecked, or from unchecked to checked.

A CheckBox control can have one of two disabled states, checked or unchecked. By default, a disabled CheckBox control displays a different background and check mark color than an enabled CheckBox control.

Creating a CheckBox control

You use the `<s:CheckBox>` tag to define a CheckBox control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block:

```
<?xml version="1.0"?>
<!-- controls\checkbox\CBSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:VGroup>
        <s:CheckBox width="100" label="Employee?"/>
    </s:VGroup>
</s:Application>
```

You can also use the `selected` property to generate a checkbox that is checked by default:

```
<?xml version="1.0"?>
<!-- controls\checkbox\CBSelected.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:VGroup>
        <s:CheckBox width="100" label="Employee?" selected="true"/>
    </s:VGroup>
</s:Application>
```

CheckBox control user interaction

When a CheckBox control is enabled and the user clicks it, the control receives focus. It displays its checked or unchecked appearance, depending on its initial state. The entire area of the CheckBox control is the click area. If the CheckBox control's text is larger than its icon, the clickable regions are above and below the icon.

If the user moves the pointer outside the area of the CheckBox control or its label while pressing the mouse button, the appearance of the CheckBox control returns to its original state and the control retains focus. The state of the CheckBox control does not change until the user releases the mouse button over the control.

Users cannot interact with a CheckBox control when it is disabled.

ColorPicker control

The ColorPicker control is part of the MX component set. There is no Spark equivalent.

The ColorPicker control lets users select a color from a drop-down swatch panel. It initially appears as a preview sample with the selected color. When a user selects the control, a color swatch panel appears. The panel includes a sample of the selected color and a color swatch panel. By default, the swatch panel displays the web-safe colors (216 colors, where each of the three primary colors has a value that is a multiple of 33, such as #CC0066).

For complete reference information, see ColorPicker in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About the ColorPicker control

When you open the ColorPicker control, the swatch panel expands over other controls on the application, and normally opens downwards. If the swatch panel would hit the lower boundary of the application, but could fit above color picker button, it opens upward.

If you set the `showTextField` property to `true` (the default), the panel includes a text box with a label for the selected color. If you display a text box and set the `editable` property to `true` (the default), the user can specify a color by entering a hexadecimal value.

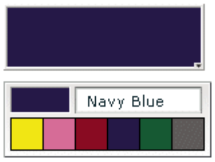
Flex populates the color swatch panel and the text box from a data provider. By default, the control uses a data provider that includes all the web-safe colors. If you use your own data provider you can specify the following:

The colors to display You *must* specify the colors if you use your own dataProvider.

Labels to display in the text box for the colors If you do not specify text labels, Flex uses the hexadecimal color values.

Additional information for each color This information can include any information that is of use to your application, such as IDs or descriptive comments.

The following image shows an expanded ColorPicker control that uses a custom data provider that includes color label values. It also uses styles to set the sizes of the display elements:



Creating a ColorPicker control

You use the `<mx:ColorPicker>` tag to define a ColorPicker control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The ColorPicker control uses a list-based data provider for the colors. For more information on this type of data provider, see “Data providers and collections” on page 898. If you omit the data provider, the control uses a default data provider with the web-safe colors. The data provider can be an array of colors or an array of objects. The following example populates a ColorPicker with a simple array of colors.

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            [Bindable]
            public var simpleDP:Array = ['0x000000', '0xFF0000', '0xFF8800',
                '0xFFFF00', '0x88FF00', '0x00FF00', '0x00FF88', '0x00FFFF',
                '0x0088FF', '0x0000FF', '0x8800FF', '0xFF00FF', '0xFFFFFF'];
        ]]>
    </fx:Script>

    <mx:ColorPicker id="cp" dataProvider="{simpleDP}"/>
</s:Application>
```

You typically use events to handle user interaction with a ColorPicker control. The following example adds an event listener for a `change` event and an `open` event to the previous example ColorPicker control:

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            //Import the event classes.
            import mx.events.DropdownEvent;
            import mx.events.ColorPickerEvent;
            [Bindable]
            public var simpleDP:Array = ['0x000000', '0xFF0000', '0xFF8800',
               '0xFFFF00', '0x88FF00', '0x00FF00', '0x00FF88', '0x00FFFF',
               '0x0088FF', '0x0000FF', '0x8800FF', '0xFF00FF', '0xFFFFFF'];

            public function openEvt(event:DropdownEvent):void {
                forChange.text="Opened";
            }
            public function changeEvt(event:ColorPickerEvent):void {
                forChange.text="Selected Item: "
                   + event.currentTarget.selectedItem + " Selected Index: "
                   + event.currentTarget.selectedIndex;
            }
        ]]>
    </fx:Script>
    <mx:VBox>
        <mx:TextArea id="forChange"
            width="150"/>
        <mx:ColorPicker id="cp"
            dataProvider="{simpleDP}"
            open="openEvt(event);"
            change="changeEvt(event);"/>
    </mx:VBox>
</s:Application>
```

The ColorPicker control dispatches open event when the swatch panel opens. It dispatches a change event when the value of the control changes due to user interaction. The currentTarget property of the object passed to the event listener contains a reference to the ColorPicker control. In this example, the event listeners use two properties of the ColorPicker control, selectedItem and selectedIndex. Every change event updates the TextArea control with the selected item and the item's index in the control, and an open event displays the word *Opened*.

If you populate the ColorPicker control from an array of color values, the target.selectedItem field contains the hexadecimal color value. If you populate it from an array of Objects, the target.selectedItem field contains a reference to the object that corresponds to the selected item.

The index of items in the ColorPicker control is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items; therefore, the target.selectedIndex value is zero-based, and a value of 2 in the preceding example refers to the data provider entry with color 0xFF8800.

Using Objects to populate a ColorPicker control

You can populate a ColorPicker control with an Array of Objects. By default, the ColorPicker uses two fields in the Objects: one named color, and another named label. The label field value determines the text in the swatch panel's text field. If the Objects do not have a label field, the control uses the color field value in the text field. You can use the ColorPicker control's colorField and labelField properties to specify different names for the color and label fields. The Objects can have additional fields, such as a color description or an internal color ID, that you can use in ActionScript.

Example: ColorPicker control that uses Objects

The following example shows a ColorPicker that uses an Array of Objects with three fields: color, label, and descript:

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPObjects.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.events.ColorPickerEvent;
            import mx.events.DropdownEvent;
            [Bindable]
            public var complexDPArray:Array = [
                {label:"Yellow", color:"0xFFFF00",
                    descript:"A bright, light color."},
                {label:"Hot Pink", color:"0xFF66CC",
                    descript:"It's HOT!"},
                {label:"Brick Red", color:"0x990000",
                    descript:"Goes well with warm colors."},
                {label:"Navy Blue", color:"0x000066",
                    descript:"The conservative favorite."},
                {label:"Forest Green", color:"0x006600",
                    descript:"Great outdoorsy look."},
                {label:"Grey", color:"0x666666",
                    descript:"An old reliable."}]
            public function openEvt(event:DropdownEvent):void {
                descriptBox.text="";
            }
            public function changeEvt(event:ColorPickerEvent):void {
                descriptBox.text=event.currentTarget.selectedItem.label
                    + ": " + event.currentTarget.selectedItem.descript;
            }
        ]]>
    </fx:Script>
    <fx:Style>
        .myStyle {
            swatchWidth:25;
            swatchHeight:25;
            textFieldWidth:95;
        }
    </fx:Style>

    <!-- Convert the Array to an ArrayCollection. Do this if
    you might change the colors in the panel dynamically. -->
    <fx:Declarations>
        <mx:ArrayCollection id="complexDP" source="{complexDPArray}"/>
    </fx:Declarations>
    <mx:VBox>
        <mx:TextArea id="descriptBox"
            width="150" height="50"/>
        <mx:ColorPicker id="cp"
            height="50" width="150"
            dataProvider="{complexDP}"
            change="changeEvt(event);"
            open="openEvt(event);"
            editable="false"/>
    </mx:VBox>
</s:Application>
```

In this example, the `selectedItem` property contains a reference to the object defining the selected item. The example uses `selectedItem.label` to access the object's `label` property (the color name), and `selectedItem.descript` to access the object's `descript` property (the color description). Every `change` event updates the `TextArea` control with the `label` property of the selected item and the item's description. The `open` event clears the current text in the `TextArea` control each time the user opens up the ColorPicker to display the swatch panel.

This example also uses several of the ColorPicker properties and styles to specify the control's behavior and appearance. The `editable` property prevents users from entering a value in the color label box (so they can only select the colors from the dataProvider). The `swatchWidth` and `swatchHeight` styles control the size of the color samples in the swatch panel. The `textFieldWidth` style ensures that the text field is long enough to accommodate the longest color name.

Using custom field names

In some cases, you might want to use custom names for the color and label fields. For example, you would use a custom name if the data comes from an external data source with custom column names. The following code changes the previous example to use custom color and label fields called cName and cVal. It also shows how to use an `<mx:dataProvider>` tag to populate the data provider:

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPCustomFieldNames.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.events.ColorPickerEvent;
            import mx.events.DropdownEvent;
            public function openEvt(event:DropdownEvent):void {
                descriptBox.text="";
            }

            public function changeEvt(event:ColorPickerEvent):void {
                descriptBox.text=event.currentTarget.selectedItem.cName
                    + ": " + event.currentTarget.selectedItem.cDescript;
            }
        ]]>
    </fx:Script>

    <fx:Style>
        .myStyle {
            swatchWidth:25;
            swatchHeight:25;
            textFieldWidth:95;
        }
    </fx:Style>

    <mx:VBox>
        <mx:TextArea id="descriptBox"
            width="150" height="50"/>
        <mx:ColorPicker id="cp"
            height="50" width="150"
            labelField="cName"
            colorField="cVal"
            change="changeEvt(event)"
            open="openEvt(event)"
            swatchPanelStyleName="myStyle"
            editable="false">
            <mx:dataProvider>
```

```
                <mx:ArrayCollection>
                    <mx:source>
                        <fx:Object cName="Yellow" cVal="0xFFFF00"
                            cDescript="A bright, light color."/>
                        <fx:Object cName="Hot Pink" cVal="0xFF66CC"
                            cDescript="It's HOT!"/>
                        <fx:Object cName="Brick Red" cVal="0x990000"
                            cDescript="Goes well with warm colors."/>
                        <fx:Object cName="Navy Blue" cVal="0x000066"
                            cDescript="The conservative favorite."/>
                        <fx:Object cName="Forest Green" cVal="0x006600"
                            cDescript="Great outdoorsy look."/>
                        <fx:Object cName="Grey" cVal="0x666666"
                            cDescript="An old reliable."/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ColorPicker>
    </mx:VBox>
</s:Application>
```

User interaction

A ColorPicker control can be editable or noneditable. In a noneditable ColorPicker control, the user must select a color from among the swatch panel options. In an editable ColorPicker control, a user can select swatch panel items or enter a hexadecimal color value directly into the label text field at the top of the swatch panel. Users can type numbers and uppercase or lowercase letters in the ranges a-f and A-F in the text box; it ignores all other non-numeric characters.

Mouse interaction

You can use the mouse to navigate and select from the control:

- Click the collapsed control to display or hide the swatch panel.
- Click any swatch in the swatch panel to select it and close the panel.
- Click outside the panel area to close the panel without making a selection.
- Click in the text field to move the text entry cursor.

Keyboard interaction

If the ColorPicker is editable and the swatch panel has the focus, alphabetic keys in the range A-F and a-f and numeric keys enter text in the color box. The Backspace and Delete keys remove text in the color text box. You can also use the following keystrokes to control the ColorPicker:

Key	Description
Control+Down Arrow	Opens the swatch panel and puts the focus on the selected swatch.
Control+Up Arrow	Closes the swatch panel, if open.
Home	Moves the selection to the first color in a row of the swatch panel. Has no effect if there is a single column.
End	Moves the selection to the last color in a row of the swatch panel. Has no effect if there is a single column.
Page Up	Moves the selection to the top color in a column of the swatch panel. Has no effect if there is a single row.
Page Down	Moves the selection to the bottom color in a column of the swatch panel. Has no effect if there is a single row.

Key	**Description**
Escape	Closes the swatch panel without changing the color in the color picker. Most Web browsers do not support using this key.
Enter	Selects the current color from the swatch panel and closes the swatch panel; equivalent to clicking a color swatch. If the focus is on the text field of an editable ColorPicker, selects the color specified by the field text.
Arrows	When the swatch panel is open, moves the focus to the next color left, right, up, and down in the swatch grid. On a single-row swatch panel, Up and Right Arrow keys are equivalent, and Down and Left Arrow keys are equivalent. On a multirow swatch panel, the selection wraps to the beginning or end of the next or previous line. On a single-row swatch panel, pressing the key past the beginning or end of the row loops around on the row. When the swatch panel is closed, but has the focus, pressing the Up and Down arrow keys has no effect. The Left and Right Arrow keys change the color picker selection, moving through the colors as if the panel were open.

***Note:** When the swatch panel is open, you cannot use the Tab and Shift+Tab keys to move the focus to another object.*

DateChooser and DateField controls

The DateChooser and DateField controls are part of the MX component set. There is no Spark equivalent.

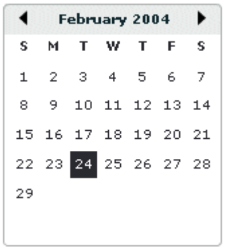
The DateChooser and DateField controls let users select dates from graphical calendars. The DateChooser control user interface is the calendar. The DateField control has a text field that uses a date chooser popup to select the date as a result. The DateField properties are a superset of the DateChooser properties.

For complete reference information, see DateChooser and DateField in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About the DateChooser control

The DateChooser control displays the name of a month, the year, and a grid of the days of the month. It contains columns labeled for the days of the week. This control is useful in applications where you want a continually visible calendar. The user can select a single date from the grid. The control contains forward and back arrow buttons to let you change the month and year. You can disable the selection of certain dates, and limit the display to a range of dates.

The following image shows a DateChooser control:



S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29						

Changing the displayed month does not change the selected date. Therefore, the currently selected date is not always visible. The DateChooser control resizes as necessary to accommodate the width of the weekday headings. Therefore, if you use day names, instead of letters, as headings, the calendar is wide enough to show the full day names.

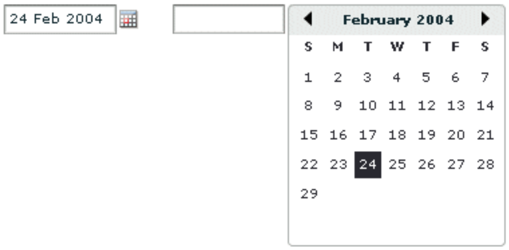
About the DateField control

The DateField control is a text field that displays the date with a calendar icon on its right side. When a user clicks anywhere inside the bounding box of the control, a date chooser that is identical to the DateChooser control pops up. If no date has been selected, the text field is blank and the current month is displayed in the date chooser.

When the date chooser is open, users can click the month scroll buttons to scroll through months and years, and select a date. When the user selects a date, the date chooser closes and the text field displays the selected date.

This control is useful in applications where you want a calendar selection tool, but want to minimize the space that the date information takes up.

The following example shows two images of a DateField control. On the left is the DateField control with the date chooser closed; the calendar icon appears on the right side of the text box. To the right is a DateField control with the date chooser open:



You can use the DateField control anywhere you want a user to select a date. For example, you can use a DateField control in a hotel reservation system, with certain dates selectable and others disabled. You can also use the DateField control in an application that displays current events, such as performances or meetings, when a user selects a date.

Creating a DateChooser or DateField control

You define a DateChooser control in MXML by using the `<mx:DateChooser>` tag. You define a DateField control in MXML by using the `<mx:DateField>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

The following example creates a DateChooser control; to create a DateField control, simply change `<mx:DateChooser>` to `<mx:DateField>`. The example uses the `change` event of the DateChooser control to display the selected date in several different formats.

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
        import mx.events.CalendarLayoutChangeEvent;

        private function useDate(eventObj:CalendarLayoutChangeEvent):void {
            // Make sure selectedDate is not null.
            if (eventObj.currentTarget.selectedDate == null) {
                return
            }
            //Access the Date object from the event object.
            day.text=eventObj.currentTarget.selectedDate.getDay();
            date.text=eventObj.currentTarget.selectedDate.getDate();
            month.text=eventObj.currentTarget.selectedDate.getMonth() + 1;
            year.text=eventObj.currentTarget.selectedDate.getFullYear();

            wholeDate.text= (eventObj.currentTarget.selectedDate.getMonth() + 1) +
                "/" + (eventObj.currentTarget.selectedDate.getDate() +
                "/" + eventObj.currentTarget.selectedDate.getFullYear());
        }
        ]]>
    </fx:Script>
    <mx:DateChooser id="date1" change="useDate(event)"/>
    <s:Form x="200">
        <s:FormItem label="Day of week">
            <s:TextInput id="day" width="100"/>
        </s:FormItem>
        <s:FormItem label="Day of month">
            <s:TextInput id="date" width="100"/>
        </s:FormItem>
        <s:FormItem label="Month">
            <s:TextInput id="month" width="100"/>
        </s:FormItem>
        <s:FormItem label="Year">
            <s:TextInput id="year" width="100"/>
        </s:FormItem>
        <s:FormItem label="Date">
            <s:TextInput id="wholeDate" width="100"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Notice that the first line of the event listener determines if the `selectedDate` property is null. This check is necessary because selecting the currently selected date while holding down the Control key deselects it, sets the `selectedDate` property to null, then dispatches the `change` event.

***Note:** The code that determines the value of the `wholeDate` field adds 1 to the month number because the DateChooser control uses a zero-based month system. In this system, January is month 0 and December is month 11.*

Using the Date class

The DateChooser and DateField controls use the `selectedDate` property to store the currently selected date, as an object of type Date. You can create Date objects to represent date and time values, or access the Date in the `selectedDate` property.

The Date class has many methods that you can use to manipulate a date. For more information on the Date class, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

In MXML, you can create and configure a Date object by using the `<mx:Date>` tag. This tag exposes the setter methods of the Date class as MXML properties so that you can initialize a Date object. For example, the following code creates a DateChooser control and sets the selected date to April 10, 2005. Notice that months are indexed starting at 0:

```
<mx:DateChooser id="date1">
        <mx:selectedDate>
            <fx:Date month="3" date="10" fullYear="2005"/>
        </mx:selectedDate>
</mx:DateChooser>
```

The following example uses inline ActionScript to set the initial selected date for a DateField control:

```
<mx:DateField id="date3" selectedDate="{new Date (2005, 3, 10)}"/>
```

You can also set the `selectedDate` property in a function, as the following example shows:

```
<fx:Script>
    <![CDATA[
        private function initDC():void {
            date2.selectedDate=new Date (2005, 3, 10);
        }
    ]]>
</fx:Script>

<mx:DateChooser id="date2" creationComplete="initDC();"/>
```

You can use property notation to access the ActionScript setter and getter methods of the `selectedDate` property Date object. For example, the following line displays the four-digit year of the selected date in a text box:

```
<s:TextInput text="{date1.selectedDate.fullYear}"/>
```

Specifying header, weekday, and today's day text styles

The following date chooser properties let you specify text styles for regions of the control:

- `headerStyleName`
- `weekDayStyleName`
- `todayStyleName`

These properties let you specify styles for the text in the header, weekday list, and today's date. You cannot use these properties to set non-text styles such as `todayColor`.

The following example defines a DateChooser control that has bold, blue header text in a 16-pixel Times New Roman font. The day-of-week headers are in bold, italic, green, 15-pixel Courier text, and today's date is bold, orange, 12-pixel Times New Roman text. Today's date background color is gray, and is set directly in the `<mx:DateChooser>` tag.

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Style>
        .myHeaderStyle{
            color:#6666CC;
            font-family:Times New Roman, Times, serif;
            font-size:16px; font-weight:bold;}
        .myTodayStyle{
            color:#CC6633;
            font-family:Times New Roman, Times, serif;
            font-size:12px; font-weight:bold;}
        .myDayStyle{
            color:#006600;
            font-family:Courier New, Courier, mono;
            font-size:15px; font-style:italic; font-weight:bold;}
    </fx:Style>
    <mx:DateChooser
        headerStyleName="myHeaderStyle"
        todayStyleName="myTodayStyle"
        todayColor="#CCCCCC"
        weekDayStyleName="myDayStyle"/>
</s:Application>
```

Specifying selectable dates

The DateChooser control has the following properties that let you specify which dates a user can select:

Property	Description
`disabledDays`	An array of days of the week that the user cannot select. Often used to disable weekend days.
`disabledRange`	An array of dates that the user cannot select. The array can contain individual Date objects, objects specifying date ranges, or both.
`selectableRange`	A single range of dates that the user can select. The user can navigate only among the months that include this range; in these months any dates outside the range are disabled. Use the `disabledRange` property to disable dates within the selectable range.

The following example shows a DateChooser control that has the following characteristics:

- The `selectableRange` property limits users to selecting dates in the range January 1 - March 15, 2006. Users can only navigate among the months of January through March 2006.
- The `disabledRanges` property prevents users from selecting January 11 or any day in the range January 23 - February 10.
- The `disabledDays` property prevents users from selecting Saturdays or Sundays.

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserSelectable.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

  <mx:DateChooser
    selectableRange="{{rangeStart: new Date(2006,0,1),
        rangeEnd: new Date(2006,2,15)}}"
    disabledRanges="{[new Date(2006,0,11),
        {rangeStart: new Date(2006,0,23), rangeEnd: new Date(2006,1,10)}]}"
    disabledDays="{[0,6]}"/>
</s:Application>
```

Setting DateChooser and DateField properties in ActionScript

Properties of the DateChooser and DateField controls take values that are scalars, Arrays, and Date objects. While you can set most of these properties in MXML, it can be easier to set some in ActionScript.

For example, the following code example uses an array to set the `disabledDays` property. Sunday (0) and Saturday (6) are disabled, which means that they cannot be selected in the calendar. This example sets the `disabledDays` property in two different ways, by using tags and by using tag attributes:

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserDisabledOption.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <!-- Use tags.-->
    <mx:DateField>
            <mx:disabledDays>
                <fx:Number>0</fx:Number>
                <fx:Number>6</fx:Number>
            </mx:disabledDays>
    </mx:DateField>
    <!-- Use tag attributes.-->
    <mx:DateField disabledDays="[0,6]"/>
</s:Application>
```

The following example sets the `dayNames`, `firstDayOfWeek`, `headerColor`, and `selectableRange` properties of a DateChooser control by using an `initialize` event:

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserInitializeEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.events.DateChooserEvent;
            private function dateChooser_init():void {
                myDC.dayNames=['Sun', 'Mon', 'Tue',
                    'Wed', 'Th', 'Fri', 'Sat'];
                myDC.firstDayOfWeek = 3;
                myDC.setStyle("headerColor", 0xff0000);
                myDC.selectableRange = {rangeStart: new Date(2009,0,1),
                    rangeEnd: new Date(2012,0,10)};
            }

            private function onScroll():void {
                myDC.setStyle("fontStyle", "italic");
            }
        ]]>
    </fx:Script>
    <mx:DateChooser id="myDC"
        width="200"
        creationComplete="dateChooser_init();"
        scroll="onScroll();"/>
</s:Application>
```

To set the `selectableRange` property, the code creates two Date objects that represent the first date and last date of the range. Users can only select dates within the specified range. This example also changes the `fontStyle` of the DateChooser control to `italics` after the first time the user scrolls it.

You can select multiple dates in a DateChooser control by using the `selectedRanges` property. This property contains an Array of objects. Each object in the Array contains two dates: a start date and an end date. By setting the dates within each object to the same date, you can select any number of individual dates in the DateChooser.

The following example uses an XML object to define the date for the DateChooser control. It then iterates over the XML object and creates an object for each date. These objects are then used to determine what dates to select in the DateChooser:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\date\ProgrammaticDateChooserSelector.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init()">
    <fx:Script>
        <![CDATA[
            private function init():void {
                dc1.displayedMonth = 1;
                dc1.displayedYear = 2008;
            }

            public function displayDates():void {
                var dateRanges:Array = [];
                for (var i:int=0; i<shows.show.length(); i++) {
                    var cDate:Date =
                        new Date(shows.show[i].showDate.toString());
                    var cDateObject:Object =
                        {rangeStart:cDate, rangeEnd:cDate};
                    dateRanges.push(cDateObject);
                }
                dc1.selectedRanges = dateRanges;
            }
        ]]>
    </fx:Script>
    <!-- Define the data for the DateChooser -->
    <fx:Declarations>
    <fx:XML id="shows" format="e4x">
        <data>
            <show>
               <showID>1</showID>
                <showDate>02/28/2008</showDate>
                <showTime>10:45am/11:15am</showTime>
            </show>
            <show>
                <showID>2</showID>
                <showDate>02/23/2008</showDate>
                <showTime>7:00pm</showTime>
            </show>
        </data>
    </fx:XML>
    </fx:Declarations>

    <mx:DateChooser id="dc1"
        showToday="false"
        creationComplete="displayDates();"/>

</s:Application>
```

Formatting dates with the DateField control

You can use the `formatString` property of the DateField control to format the string in the control’s text field. The `formatString` property can contain any combination of "MM", "DD", "YY", “YYYY”, delimiter, and punctuation characters. The default value is "MM/DD/YYYY".

In the following example, you select a value for the `formatString` property from the drop-down list:

```
<?xml version="1.0"?>
<!-- controls\date\DateFieldFormat.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <mx:HBox>
        <mx:ComboBox id="cb1">
            <mx:ArrayCollection>
                <fx:String>MM/DD/YY</fx:String>
                <fx:String>MM/DD/YYYY</fx:String>
                <fx:String>DD/MM/YY</fx:String>
                <fx:String>DD/MM/YYYY</fx:String>
                <fx:String>DD MM, YYYY</fx:String>
            </mx:ArrayCollection>
        </mx:ComboBox>

        <mx:DateField id="date2"
            editable="true"
            width="100"
            formatString="{cb1.selectedItem}"/>
    </mx:HBox>
</s:Application>
```

The DateField control also lets you specify a formatter function. A formatter function converts the date to a string in your preferred format for display in the control's text field. The DateField `labelFunction` property and the Spark DateTimeFormatter class help you format dates.

By default, the date in the DateField control text field is formatted in the form "MM/DD/YYYY". You use the `labelFunction` property of the DateField control to specify a function to format the date displayed in the text field and return a String containing the date. The function has the following signature:

```
public function formatDate(currentDate:Date):String {
    ...
    return dateString;
}
```

You can choose a different name for the function, but it must take a single argument of type Date and return the date as a String for display in the text field. The following example defines the function `formatDate()` to display the date in the form dd-mm-yyyy, such as 03-12-2011. This function uses a Spark DateTimeFormatter object to do the formatting:

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            private function formatDateTime(date:Date):String {
                return dtf.format(date);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <s:DateTimeFormatter id="dtf" dateTimePattern="MM-dd-yyyy"/>
    </fx:Declarations>

    <mx:DateField id="df" labelFunction="formatDateTime" parseFunction="{null}"/>
</s:Application>
```

The `parseFunction` property specifies a function that parses the date entered as text in the text field of the DateField control and returns a Date object to the control. If you do not allow the user to enter a date in the text field, set the `parseFunction` property to `null` when you set the `labelFunction` property.

If you want to let the user enter a date in the control's text field, specify a function to the `parseFunction` property that converts the text string to a Date object for use by the DateField control. If you set the `parseFunction` property, it should typically perform the reverse of the function specified to the `labelFunction` property.

The function specified to the `parseFunction` property has the following signature:

```
public function parseDate(valueString:String, inputFormat:String):Date {
    ...
    return newDate
}
```

Where the `valueString` argument contains the text string entered by the user in the text field, and the `inputFormat` argument contains the format of the string. For example, if you only allow the user to enter a text string by using two characters for month, day, and year, then pass "MM/DD/YY" to the `inputFormat` argument.

User interaction

The date chooser includes arrow buttons that let users move between months. Users can select a date with the mouse by clicking the desired date.

Clicking a forward month arrow advances a month; clicking the back arrow displays the previous month. Clicking forward a month on December, or back on January, moves to the next (or previous) year. Clicking a date selects it. By default, the selected date is indicated by a green background around the date and the current day is indicated by a black background with the date in white. Clicking the currently selected date deselects it.

The following keystrokes let users navigate DateChooser and DateField controls:

Key	**Use**
Left Arrow	Moves the selected date to the previous enabled day in the month. Does not move to the previous month. Use the Shift key plus the Left Arrow key to access disabled days.
Right Arrow	Moves the selected date to the next enabled day in the month. Does not move to the next month. Use the Shift key plus the Right Arrow key to access disabled days.
Up Arrow	Moves the selected date up the current day of week column to the previous enabled day. Does not move to the previous month. Use the Shift key plus the Up Arrow key to access disabled days.
Down Arrow	Moves the selected date down the current day of week column to next enabled day. Does not move to the next month. Use the Shift key plus the Down Arrow key to access disabled days.
Page Up	Displays the calendar for the previous month.
Page Down	Displays the calendar for the next month.
Home	Moves the selection to the first enabled day of the month.
End	Moves the selection to the last enabled day of the month.
+	Move to the next year.
-	Move to the previous year.
Control+Down Arrow	DateField only: open the DateChooser control.
Control+Up Arrow	DateField only: close the DateChooser control.
Escape	DateField only: cancel operation.
Enter	DateField only: selects the date and closes the DateChooser control.

***Note:** The user must select the control before using navigation keystrokes. In a DateField control, all listed keystrokes work only when the date chooser is displayed.*

Image control

Adobe Flex supports several image formats, including GIF, JPEG, and PNG. You can import these images into your applications by using the Spark Image control or BitmapImage. To load SWF files, you use the SWFLoader control.

The Image control is part of both MX and Spark component sets. While you can use the MX Image control in your application, Adobe recommends that you use the Spark Image control instead.

Flex provides the following image controls:

Image control	**Use**
BimapImage	BitmapImage is a light-weight image control that can load both local and remote assets. You typically use the BitmapImage when you don't require a broken image icon or a preloader. The BitmapImage also supports runtime loading of images. However, when you load images at runtime, you should be aware of the security restrictions of Flash Player or AIR. For more information, see "Security" on page 117.
Spark Image	The Spark Image is a skinnable component that leverages a BitmapImage class as its main skin part. The Spark Image control provides a preloader as well as a broken image icon to indicate an invalid image state. The image loader and the broken-image icon are both customizable. For example, you can provide a broken image icon representing an invalid URL, or provide a progress bar as the image loads. The Spark Image control also provides customizable skinning for borders, frames, and loading states. For more information, see "Skinning the Spark Image control" on page 684. The Spark Image control lets you load local and trusted assets, as well as remote and untrusted assets. Security restrictions are applicable in case of untrusted assets. For more information, see "Loading images using a customizable load interface" on page 687.
SWFLoader	Flex also includes the SWFLoader control for loading applications built in Flex. You typically use the Image control for loading static graphic files, and use the SWFLoader control for loading SWF files and applications built in Flex. The Image control is also designed for use in custom item renderers and item editors. For more information on the SWFLoader control, see "SWFLoader control" on page 715.

The following example shows a BitmapImage embedded at compile time and a BitmapImage loaded at runtime:

```
<?xml version="1.0"?>
<!-- controls\image\ImageBitmapImageExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="700">

    <s:Panel title="BitmapImage Examples" width="600">
        <s:layout>
            <s:HorizontalLayout horizontalAlign="center"/>
        </s:layout>
        <s:BorderContainer width="50%" height="100%">
            <s:layout>
                <s:VerticalLayout
                   verticalAlign="middle"
                   horizontalAlign="center"/>
            </s:layout>
            <s:Label text="BitmapImage embedded at compile time"/>
            <s:BitmapImage id="embimg" source="@Embed(source='fblogo.jpg')"
               height="50" width="50"/>
        </s:BorderContainer>
        <s:BorderContainer width="50%" height="100%">
            <s:layout>
                <s:VerticalLayout
                   verticalAlign="middle"
                   horizontalAlign="center"/>
            </s:layout>
            <s:Label text="BitmapImage loaded at runtime"/>
            <s:BitmapImage id="runtimeimg" source="../assets/flexlogo.jpg"
               height="50" width="50"/>
        </s:BorderContainer>
    </s:Panel>
</s:Application>
```

For a video on using Spark Image and BitmapImage, see www.adobe.com/go/learn_flexsparkimagevideo_en

About loading images

Flex supports loading GIF, JPEG, and PNG files at runtime, and also embedding GIF, JPEG, and PNG files at compile time. The method you choose depends on the file types of your images and your application parameters.

Embedded images load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded images also require you to recompile your applications whenever your image files change. For an overview of resource embedding, see "Embedding assets" on page 1699.

The alternative to embedding a resource is to load the resource at runtime. You can load a resource from the local file system in which the SWF file runs. You can also access a remote resource, typically though an HTTP request over a network. These images are independent of your application built with Flex, so you can change them without causing a recompile operation as long as the names of the modified images remain the same. The referenced images add no additional overhead to an application's initial loading time. However, you can sometimes experience a delay when you use the images and load them into Adobe Flash Player or AIR.

A SWF file can access one type of external resource only, either local or over a network; it cannot access both types. You determine the type of access allowed by the SWF file by using the `use-network` flag when you compile your application. When `use-network` flag is set to `false`, you can access resources in the local file system, but not over the network. The default value is `true`, which allows you to access resources over the network, but not in the local file system.

For more information on the `use-network` flag, see "Flex compilers" on page 2164.

When you load images at runtime, you should be aware of the security restrictions of Flash Player or AIR. For example, you can reference an image by using a URL, but the default security settings only permit applications built in Flex to access resources stored on the same domain as your application. To access images on other servers, ensure that the domain that you are loading from uses a crossdomain.xml file.

For more information on application security, see "Security" on page 117.

Loading and embedding images with Spark Image and BitmapImage controls

The Spark Image control and BitmapImage support the following actions when you load an image:

Specifying the image path

The value of the `source` property of a Spark Image control or BitmapImage specifies a relative or absolute path, or URL to the imported image file. If the value is relative, it is relative to the directory that contains the file that uses the tag.

The `source` property has the following forms:

1 `source="@Embed(source='relativeOrAbsolutePath')"`

   The referenced image is packaged within the generated SWF file at compile time when Flex creates the SWF file for your application. You can embed GIF, JPEG, and PNG files. When embedding an image, the value of the `source` property must be a relative or absolute path to a file on the Flex developer's local file system; it cannot be a URL.

   The following example embeds a JPEG image into an application that is built in Flex:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleEmbed.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Image id="loader1" source="@Embed(source='logo.jpg')"/>
</s:Application>
```

   In this example, the size of the image is the default size of the image file.

2 `source="relativeOrAbsolutePathOrURL"`

   Flex loads the referenced image file at runtime, typically from a location that is deployed on a web server; the image is not packaged as part of the generated SWF file. You can only reference GIF, JPEG, and PNG files.

   The following example accesses a JPEG image at runtime:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:Image id="loader1" source="assets/flexlogo.jpg"/>
</s:Application>
```

Because you did not use @Embed in the source property, Flex loads the image at runtime. In this case, the location is the same directory as the page that embedded the SWF file.

When the use-network flag is set to false, you can access resources in the local file system, but not over the network. The default value is true, which allows you to access resources over the network, but not in the local file system.

In many applications, you create a directory to hold your application images. Commonly, that directory is a subdirectory of your main application directory. The source property supports relative paths to images, which let you specify the location of an image file relative to your application directory.

The following example stores all images in an assets subdirectory of the application directory:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleAssetsDir.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Image id="loader1" source="@Embed(source='assets/logo.jpg')"/>
</s:Application>
```

The following example uses a relative path to reference an image in an assets directory at the same level as the application's root directory:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleAssetsDirTop.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Image id="loader1" source="@Embed(source='../assets/logo.jpg')"/>
</s:Application>
```

You can also reference an image by using a URL, but the default security settings only permit applications to access resources stored on the same domain as your application. To access images on other servers, make sure that the domain that you are loading the image from uses a crossdomain.xml file that allows cross-domain access.

Cross-domain image loading is supported with restrictions. You can load local and trusted assets, as well as remote and untrusted assets. When you use untrusted assets, security restrictions are applicable. When a cross-domain image is loaded from an untrusted source, the security policy does not allow access to the image content. Then, the trustedSource property of the BimapImage and Spark Image is set to false, and the following image properties do not function:

- BitmapFillMode.REPEAT
- scaleGrid
- smooth
- smoothingQuality

Any advanced operations on the image data, such as, high-quality scaling or tiling, are not supported.

For more information, see "Security" on page 117.

The following example shows how to reference an image by using a URL:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleAssetsURL.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Image id="image1"
        source="http://localhost:8100/flex/assets/logo.jpg"/>
</s:Application>
```

***Note:** You can use relative URLs for images hosted on the same web server as the application, but load these images over the Internet rather than access them locally.*

Using an image multiple times

You can use the same image multiple times in your application by using the normal image import syntax each time. Flex only loads the image once, and then references the loaded image as many times as necessary.

Sizing and scaling an image

You can size and scale images using both the Spark Image and BimapImage controls.

Sizing an image

Flex sets the height and width of an imported image to the height and width settings in the image file. By default, Flex does not resize the image.

To set an explicit height or width for an imported image, set its `height` and `width` properties of the image control. Setting the `height` or `width` property prevents the parent from resizing it. The `scaleContent` property has a default value of `true`; therefore, Flex scales the image as it resizes it to fit the specified height and width. The aspect ratio is maintained by default for a Spark Image, so the image may not completely fill the designated space. Set the `scaleContent` property to `false` to disable scaling.

When you resize a BitmapImage, the aspect ratio is not maintained, by default. To ensure that you maintain the aspect ratio, change the value of the `scaleMode` property from `stretch` to `letterbox` or `zoom`. For more information about image aspect ratios, see below.

One common use for resizing an image is to create image thumbnails. In the following example, the image has an original height and width of 100 by 100 pixels. By specifying a height and width of 20 by 20 pixels, you create a thumbnail of the image.

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleThumbnail.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:BitmapImage id="image1"
        source="@Embed(source='logo.jpg')"
        width="20" height="20"
        smooth="true" smoothingQuality="high"/>
    <s:BitmapImage id="image2"
        source="@Embed(source='logo.jpg')"/>
</s:Application>
```

When you downscale a BimapImage, the default quality for scaled bitmaps, `BitmapSmoothingQuality.DEFAULT`, is used.

If you want to preserve the aspect ratio when creating image thumbnails, and achieve the best quality result when downscaling a BitmapImage, set the `smoothingQuality` property to `BitmapSmoothingQuality.HIGH`. The `smoothingQuality` property is applicable only when you set the `smooth` property of the BitmaImage to `true`. When you do so, a multi-bit-rate step algorithm is applied to the image, resulting in a much higher quality downscale than the default. This option is useful in creating high-quality image thumbnails.

To let Flex resize the image as part of laying out your application, set the `height` and `width` properties to a percentage value. Flex attempts to resize components with percentage values for these properties to the specified percentage of their parent container. You can also use the `maxHeight` and `maxWidth` and `minHeight` and `minWidth` properties to limit resizing. For more information on resizing, see "Introduction to containers" on page 326.

Maintaining aspect ratio when sizing

The aspect ratio of an image is the ratio of its width to its height. For example, a standard NTSC television set uses an aspect ratio of 4:3, and an HDTV set uses an aspect ratio of 16:9. A computer monitor with a resolution of 640 by 480 pixels also has an aspect ratio of 4:3. A square has an aspect ratio of 1:1.

All images have an inherent aspect ratio. When you use the `height` and `width` properties of the Image control to resize an image, Flex preserves the aspect ratio of the image, by default, so that it does not appear distorted.

By preserving the aspect ratio of the image, Flex might not draw the image to fill the entire height and width specified for the `<s:Image>` tag. For example, if your original image is a square 100 by 100 pixels, which means it has an aspect ratio of 1:1, and you use the following statement to load the image:

```
<s:Image source="myImage.jpg" height="200" width="200"/>
```

The image increases to four times its original size and fills the entire 200 x 200 pixel area.

The following example sets the height and width of the same image to 150 by 200 pixels, an aspect ratio of 3:4:

```
<s:Image source="myImage.jpg" height="150" width="200"/>
```

In this example, you do not specify a square area for the resized image. Flex maintains the aspect ratio of a Spark Image by default because the `scaleMode` property is set to `letterbox`. Therefore, Flex sizes the image to 150 by 150 pixels, the largest possible image that maintains the aspect ratio and conforms to the size constraints. The other 50 by 150 pixels remain empty. However, the `<s:Image>` tag reserves the empty pixels and makes them unavailable to other controls and layout elements.

To ensure that you maintain the aspect ratio when you resize a BitmapImage, change the value of the `scaleMode` property from `stretch` to `letterbox` or `zoom`.

When you resize an image, you can apply the required alignment properties. The Spark Image and BitmapImage both have `horizontalAlign` and `verticalAlign` properties that you can specify, as required.

By default, the `horizontalAlign` and `verticalAlign` properties are set to `center`. For example, when you resize an image that has an original height and width of 1024 by 768 to a height and width of 100 by 100, there is extra space at the top and bottom of the image. You can set the the `horizontalAlign` and `verticalAlign` properties as required to overcome the extra space.

You can also use a Resize effect to change the width and height of an image in response to a trigger. As part of configuring the Resize effect, you specify a new height and width for the image. Flex maintains the aspect ratio of the image by default, so it resizes the image as much as possible to conform to the new size, while maintaining the aspect ratio. For example, place your pointer over the image in this example to enlarge it, and then move the mouse off the image to shrink it to its original size:

```
<?xml version="1.0"?>
<!-- controls\image\ImageResize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.effects.Resize;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <s:Resize id="resizeBig"
            widthFrom="22" widthTo="44"
            heightFrom="27" heightTo="54"/>
        <s:Resize id="resizeSmall"
            widthFrom="44" widthTo="22"
            heightFrom="54" heightTo="27"/>
    </fx:Declarations>
    <s:Image width="22" height="27"
        source="@Embed('logo.jpg')"
        rollOverEffect="{resizeBig}"
        rollOutEffect="{resizeSmall}"/>

</s:Application>
```

For more information on the Resize effect, see “Introduction to effects” on page 1784.

Scaling images using fillMode and scaleMode properties

You use the `fillMode` image property to determine how an image fills into a region.

You can use the `fillMode` property in a tag or in ActionScript, and set one of the following values:

- `scale`: scales the image to fill the bounding region.
- `clip`: displays the image in its original size. If the image is larger than the bounding region, the image is clipped to the size of the bounding region. If the image is smaller than the bounding region, an empty space is left.
- `repeat`: repeats or tiles the image and fills into the dimensions of the bounding region.

When you set the `fillMode` property to `scale` , you can set the `scaleMode` property to `stretch`, `letterbox`, or `zoom`.

When you use the `fillMode` property in ActionScript, you use `BitmapFillMode.CLIP`, `BitmapFillMode.REPEAT`, or `BitmapFillMode.SCALE`.

To stretch the image content to fit the bounding region, set the attribute value to `BitmapScaleMode.STRETCH`. If you set the value to `BitmapScaleMode.LETTERBOX`, the image content is sized to fit the region boundaries while maintaining the same aspect ratio as the original unscaled image. For `BitmapScaleMode.ZOOM`, the image content is scaled to fit with respect to the original unscaled image's aspect ratio. This results in cropping along one axis.

The following example shows the Flex logo scaled in the letterbox-mode and stretch-mode using the `fillMode` and `scaleMode` properties:

```
<?xml version="1.0"?>
<!-- controls\image\ImageScaling.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:controlBarContent>
        <s:Form>
            <s:FormItem label="scaleMode:">
                <s:DropDownList id="ddl" selectedItem="letterbox">
                    <s:dataProvider>
                        <s:ArrayList source="[letterbox,scale]" />
                    </s:dataProvider>
                </s:DropDownList>
            </s:FormItem>
        </s:Form>
    </s:controlBarContent>

    <s:Image id="image"
       source="assets/flexlogo.jpg"
       height="20" width="20"
       fillMode="scale"
       scaleMode="{ddl.selectedItem}"
       left="20" right="20" top="20" bottom="20" />
</s:Application>
```

For more code examples, see Peter deHaan's blog.

Skinning the Spark Image control

The Spark Image control lets you customize the appearance of you image through skinning.

When you use the `<s:Image>` tag to import an image, a default skin class, ImageSkin, is created. The default skin provides a simple image skin that has a content loader with a generic progress bar and a broken image icon to reflect invalid content.

The ImageSkin class provides Spark and wireframe-themed skins that let you customize the content loader and the icon to identify images that failed to load. You also have customizable skinning for borders, frames, and loading states.

The ImageSkin class defines the following optional skin parts:

- `imageDisplay:BitmapImage`: Lets you specify the image to display for invalid content or loading error.
- `ProgressIndicator:Range`: Lets you specify the range, maximum, and minimum values of the progress bar. Use the `stepSize` property to specify the amount by which the value changes when the `changeValueByStep()` method is called.

For information about the different skin states, see the source code for the ImageSkin class.

For examples on customizing skins and icons for the Spark Image control, see Peter deHann’s blog post.

Caching and queuing of images

The ContentCache class supports caching and queuing of remote image assets, and can be associated with the BitmapImage or Spark Image controls.

You set the caching and queuing properties for the image content loader as follows:

```
<fx:Declarations>
    <s:ContentCache id="imgcache" enableCaching="true" enableQueueing="true"
    maxActiveRequests="1" maxCacheEntries="10"/>
</fx:Declarations>
```

To have a content loader with only queuing capabilities and no caching capabilities, set `enableCaching` to `false`.

The difference that you notice in image-loading when you use the caching-queuing mechanism can be significant, especially with large images. Without caching, large images can take several seconds to load. For example, you can use caching and queuing to prevent image-flickering in a scrolling list.

The ContentCache class provides the following methods and properties to manage caching and queuing:

Caching properties

Property name	**Description**
`maxCacheEntries`	Specifies the number of cache entries that can be stored at a given time. When the cache entries exceed the specified number, the least recent cache entries are automatically removed. The default value is 100.
`enableQueuing`	Enables queuing of images. The default value is `false`.
`enableCaching`	Enables caching of images. The default value is `true`.
`maxActiveRequests`	Specifies the number of pending load requests that can be active at a given time when image-queuing is enabled. The default value is 2.

Caching methods

Method name	**Description**
`load()`	Returns the requested image using the `ContentRequest` instance. If the image is still loading, `ContentRequest` returns a value of `false`. If the requested image is found in the cache, `ContentRequest` returns a value of `complete`.

`removeAllCacheEntries()`
`removeCacheEntry()`
	Removes cache entries.
	Use `removeCacheEntry()` to remove a single cache entry that's no longer needed. Use `removeAllCacheEntries()` to remove all the cached content.

`getCacheEntry()`	Checks for a pre-existing cache entry, and if found, obtains a reference to the cache entry.
`addCacheEntry()`	Adds a new cache entry. Use `addCacheEntry` to manually add a cache entry.

Queuing methods

Method	**Description**
`prioritize()`	Forces all pending requests of the specified contentLoaderGrouping to be moved to the top of the request list, when image-queuing is enabled. Any active URL requests at that time are canceled and requeued if their contentLoaderGrouping does not match the prioritized content grouping.
`removeAllQueueEntries()`	Cancels all the queued requests.

Example on caching and queuing images

The following example shows loading of images using the contentLoader. Caching and queuing are enabled and custom item renderers are used.

```
<?xml version="1.0"?>
<!-- controls\image\ImageCaching.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";

        s|Image {
            enableLoadingState: true;
        }
    </fx:Style>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;

            protected var arrList:ArrayList = new ArrayList();

            protected function init():void {
                var rand:String = new Date().time.toString()
                var arr:Array = [];
                arr.push({src:"http://localhost:8100/flex/assets/fblogo.jpg?" + rand,
cache:ldr});
                arr.push({src:"http://localhost:8100/flex/assets/flexlogo.jpg?" + rand,
cache:ldr});

                arrList = new ArrayList(arr);
                dataGrp.dataProvider = arrList;
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:ContentCache id="ldr" enableQueueing="true"
           maxActiveRequests="1" maxCacheEntries="10"/>
    </fx:Declarations>
    <s:Panel title="Loading images using the caching-queuing mechanism" x="20" y="20">
        <s:controlBarContent>
            <s:Button label="Load Images" click="init();"/>
        </s:controlBarContent>
        <s:DataGroup id="dataGrp">
            <s:layout>
                <s:TileLayout />
            </s:layout>
            <s:itemRenderer>
                <fx:Component>
                    <s:ItemRenderer dataChange="imageDisplay.source = data.src;">
                        <s:Image id="imageDisplay" contentLoaderGrouping="gr1"
                           contentLoader="{data.cache}" width="200" height="200" />
                    </s:ItemRenderer>
                </fx:Component>
            </s:itemRenderer>
        </s:DataGroup>
    </s:Panel>
</s:Application>
```

***Note:** You can replace the URLs in the example to access images over the Internet rather than access them locally.*

Loading images using a customizable load interface

Flex provides a customizable content loader, IContentLoader, to load images. The IContentLoader has a simple interface that contains a `load` method to manage the load requests. For a given source, which is usually a URL, the `load` method returns the requested resources using the `ContentRequest`instance. The `ContentRequest` instance provides the load progress information and also notifies any content validation errors.

The `load` method also provides a `contentLoaderGrouping` instance to manage multiple load requests as a group. For example, the ContentCache's queuing methods allows requests to be prioritized by `contentLoaderGrouping`.

HRule and VRule controls

The HRule and VRule controls are part of the MX component set. There is no Spark equivalent.

The HRule (Horizontal Rule) control creates a single horizontal line and the VRule (Vertical Rule) control creates a single vertical line. You typically use these controls to create dividing lines within a container.

Creating HRule and VRule controls

You define HRule and VRule controls in MXML by using the `<mx:HRule>` and `<mx:VRule>` tags, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\rule\RuleSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:VGroup>
        <mx:Label text="Above"/>
        <mx:HRule/>
        <mx:Label text="Below"/>
        <s:HGroup>
            <mx:Label text="Left"/>
            <mx:VRule/>
            <mx:Label text="Right"/>
        </s:HGroup>
    </s:VGroup>

</s:Application>
```

You can also use properties of the HRule and VRule controls to specify line width, stroke color, and shadow color, as the following example shows:

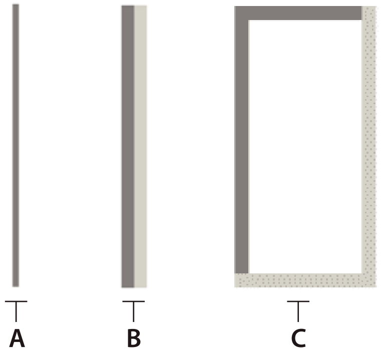
```
<?xml version="1.0"?>
<!-- controls\rule\RuleProps.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:VGroup>
        <mx:Label text="Above"/>
        <mx:HRule shadowColor="0xff0000"/>
        <mx:Label text="Below"/>
        <s:HGroup>
            <mx:Label text="Left"/>
            <mx:VRule strokeWidth="10" strokeColor="0x00ff00"/>
            <mx:Label text="Right"/>
        </s:HGroup>
    </s:VGroup>
</s:Application>
```

Sizing HRule and VRule controls

For the HRule and VRule controls, the `strokeWidth` property determines how Flex draws the line, as follows:

- If you set the `strokeWidth` property to 1, Flex draws a 1-pixel-wide line.
- If you set the `strokeWidth` property to 2, Flex draws the rule as two adjacent 1-pixel-wide lines, horizontal for an HRule control or vertical for a VRule control. This is the default value.
- If you set the `strokeWidth` property to a value greater than 2, Flex draws the rule as a hollow rectangle with 1-pixel-wide edges.

The following example shows all three options:



***A.** strokeWidth = 1* ***B.** default strokeWidth = 2* ***C.** strokeWidth = 10*

If you set the `height` property of an HRule control to a value greater than the `strokeWidth` property, Flex draws the rule within a rectangle of the specified height, and centers the rule vertically within the rectangle. The height of the rule is the height specified by the `strokeWidth` property.

If you set the `width` property of a VRule control to a value greater than the `strokeWidth` property, Flex draws the rule within a rectangle of the specified width, and centers the rule horizontally within the rectangle. The width of the rule is the width specified by the `strokeWidth` property.

If you set the `height` property of an HRule control or the `width` property of a VRule control to a value smaller than the `strokeWidth` property, the rule is drawn as if it had a `strokeWidth` property equal to the `height` or `width` property.

***Note:** If the `height` and `width` properties are specified as percentage values, the actual pixel values are calculated before the `height` and `width` properties are compared to the `strokeWidth` property.*

The `strokeColor` and `shadowColor` properties determine the colors of the HRule and VRule controls. The `strokeColor` property specifies the color of the line as follows:

- If you set the `strokeWidth` property to 1, specifies the color of the entire line.
- If you set the `strokeWidth` property to 2, specifies the color of the top line for an HRule control, or the left line for a VRule control.
- If you set the `strokeWidth` property to a value greater than 2, specifies the color of the top and left edges of the rectangle.

The `shadowColor` property specifies the shadow color of the line as follows:

- If you set the `strokeWidth` property to 1, does nothing.
- If you set the `strokeWidth` property to 2, specifies the color of the bottom line for an HRule control, or the right line for a VRule control.
- If you set the `strokeWidth` property to a value greater than 2, specifies the color of the bottom and right edges of the rectangle.

Setting style properties

The `strokeWidth`, `strokeColor`, and `shadowColor` properties are style properties. Therefore, you can set them in MXML as part of the tag definition, set them by using the `<fx:Style>` tag in MXML, or set them by using the `setStyle()` method in ActionScript.

The following example uses the `<fx:Style>` tag to set the default value of the `strokeColor` property of all HRule controls to `#00FF00` (lime green), and the default value of the `shadowColor` property to `#0000FF` (blue). This example also defines a class selector, called `thickRule`, with a `strokeWidth` of 5 that you can use with any instance of an HRule control or VRule control:

```
<?xml version="1.0"?>
<!-- controls\rule\RuleStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";

        .thickRule {strokeWidth:5}
        mx|HRule {strokeColor:#00FF00; shadowColor:#0000FF}
    </fx:Style>
    <mx:HRule styleName="thickRule"/>
</s:Application>
```

HSlider and VSlider controls

The HSlider and VSlider controls are part of both the MX and Spark component sets. While you can use the MX HSlider and VSlider controls in your application, Adobe recommends that you use the Spark controls instead.

You can use the slider controls to select a value by moving a slider thumb between the end points of the slider track. The current value of the slider is determined by the relative location of the thumb between the end points of the slider. The slider end points correspond to the slider’s minimum and maximum values.

By default, the minimum value of a slider is 0 and the maximum value is 10. The current value of the slider can be any value in a continuous range between the minimum and maximum values. It can also be one of a set of discrete values, depending on how you configure the control.

About slider controls

Flex provides two slider controls: the HSlider (Horizontal Slider) control, which creates a horizontal slider, and the VSlider (Vertical Slider) control, which creates a vertical slider. The slider controls contain a track and a slider thumb. You can optionally show or hide tooltips and data tips, which show the data value as the user drags the slider thumb. The slider controls do not contain a label property, but you can add labels to the component skin.

The following code example shows a horizontal slider and a vertical slider with different properties set. In the horizontal slider, the values increment and decrement by a value of 0.25. In the vertical slider, the data tip is hidden but a tool tip is visible.

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:Group>
        <s:layout>
            <s:HorizontalLayout paddingTop="10"
                paddingLeft="10"/>
            </s:layout>
        <s:HSlider id="horizontalBar"
            snapInterval=".25"/>
        <s:VSlider id="volumeBar"
            showDataTip="false"
            toolTip="Volume"/>
    </s:Group>
</s:Application>
```

Creating a slider control

You define an HSlider control in MXML by using the `<s:HSlider>` tag. Define a VSlider control by using the `<s:VSlider>` tag. Specify an `id` value if you intend to refer to a component elsewhere, either in another tag or in an ActionScript block.

You can bind the `value` property of a slider to another control to display the current value of the slider. The following example binds the `value` property to a Label box. Because the `liveDragging` style property is set to `true`, the value in the text box updates as the slider moves. The `dataTipPrecision` property is set to 0 to show whole numbers.

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Group>
        <s:layout>
            <s:HorizontalLayout paddingTop="10"
                paddingLeft="10"/>
            </s:layout>
        <s:HSlider id="mySlider"
            liveDragging="true"
            dataTipPrecision="0"/>
        <s:Label text="The slider value is: {Math.round(mySlider.value)}"/>
    </s:Group>

</s:Application>
```

Using slider events

The slider controls let the user select a value by moving the slider thumb between the minimum and maximum values of the slider. You typically use events with the slider to recognize when the user has moved the thumb and to determine the current value associated with the slider.

The slider controls can dispatch the events described in the following table:

Event	Description
change	Dispatches when the user moves the thumb. If the liveDragging property is true, the event is dispatched continuously as the user moves the thumb. If liveDragging is false, the event is dispatched when the user releases the slider thumb.
thumbDrag	Dispatches when the user moves a thumb.
thumbPress	Dispatches when the user selects a thumb by using the pointer.
thumbRelease	Dispatches when the user releases the pointer after a thumbPress event occurs.

The following code example uses a change event to enable a Submit button when the user releases the slider thumb:

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            public function changeRB():void {
            myButton.enabled=true;
            }
        ]]>
    </fx:Script>
    <s:Group>
        <s:layout>
            <s:VerticalLayout paddingTop="10"
            paddingLeft="10"/>
        </s:layout>
        <s:HSlider id="mySlider"
            thumbRelease="changeRB()"/>
        <s:Button id="myButton" enabled="false" label="Submit"/>

    </s:Group>
</s:Application>
```

By default, the liveDragging property of the slider control is set to false. A value of false means that the control dispatches the change event when the user releases the slider thumb. If you set liveDragging to true, the control dispatches the change event continuously as the user moves the thumb, as shown in “Creating a slider control” on page 691.

Using data tips

By default, when you select a slider thumb, a data tip appears, showing the current value of the slider. As you move the selected thumb, the data tip shows the new slider value. You can disable data tips by setting the showDataTip property to false.

You can use the dataTipFormatFunction property to specify a callback function to format the data tip text. This function takes a single String argument containing the data tip text. It returns a String containing the new data tip text, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderDataTip -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            private function myDataTipFunc(val:String):String {
                return "Current value: " + String(val);
            }
        ]]>
    </fx:Script>
    <s:layout>
        <s:HorizontalLayout paddingTop="10"
            paddingLeft="10"/>
    </s:layout>
    <s:HSlider
        height="80"
        dataTipFormatFunction="myDataTipFunc"/>
</s:Application>
```

In this example, the data tip function prefixes the data tip text with the String "Current value:". You can modify this example to insert other characters, such as a dollar sign ($), on the data tip.

Keyboard navigation

The HSlider and VSlider controls have the following keyboard navigation features when the slider control has focus:

Key	Description
Left Arrow	Decrement the value of an HSlider control by one value interval or, if you do not specify a value interval, by one pixel.
Right Arrow	Increment the value of a HSlider control by one value interval or, if you do not specify a value interval, by one pixel.
Home	Moves the thumb of an HSlider control to its minimum value.
End	Moves the thumb of an HSlider control to its maximum value.
Up Arrow	Increment the value of an VSlider control by one value interval or, if you do not specify a value interval, by one pixel.
Down Arrow	Decrement the value of a VSlider control by one value interval or, if you do not specify a value interval, by one pixel.

LinkBar control

The LinkBar control is part of the MX component set. There is no Spark equivalent.

A LinkBar control defines a horizontal or vertical row of LinkButton controls that designate a series of link destinations. You typically use a LinkBar control to control the active child container of a ViewStack container, or to create a standalone set of links.

Creating a LinkBar control

One of the most common uses of a LinkBar control is to control the active child of a ViewStack container. For an example, see “MX ViewStack navigator container” on page 630.

You can also use a LinkBar control on its own to create a set of links in your application. In the following example, you define an itemClick handler for the LinkBar control to respond to user input, and use the `dataProvider` property of the LinkBar to specify its label text:

```
<?xml version="1.0"?>
<!-- controls\bar\LBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:LinkBar borderStyle="solid"
        itemClick="navigateToURL(new URLRequest('http://www.adobe.com/' +
            String(event.label).toLowerCase()), '_blank');">
        <mx:dataProvider>
            <fx:String>Flash</fx:String>
            <fx:String>Director</fx:String>
            <fx:String>Dreamweaver</fx:String>
            <fx:String>ColdFusion</fx:String>
        </mx:dataProvider>
    </mx:LinkBar>
</s:Application>
```

In this example, you use the `<mx:dataProvider>` tag to define the label text. The event object passed to the `itemClick` handler contains the label selected by the user. The handler for the `itemClick` event constructs an HTTP request to the Adobe website based on the label, and opens that page in a new browser window.

You can also bind data to the `dataProvider` property to populate the LinkBar control, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\LBarBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var linkData:ArrayCollection = new ArrayCollection([
                {label:'Flash', url:'http://www.adobe.com/products/flash/'},
                {label:'Director', url:'http://www.adobe.com/products/director/'},
                {label:'Dreamweaver', url:'http://www.adobe.com/products/dreamweaver/'},
                {label:'ColdFusion', url:'http://www.adobe.com/products/coldfusion/'}]);
        ]]>
    </fx:Script>

    <mx:LinkBar
        dataProvider="{linkData}"
        horizontalAlign="right"
        borderStyle="solid"
        itemClick="navigateToURL(new URLRequest(event.item.url), '_blank');">
    </mx:LinkBar>
</s:Application>
```

A LinkBar control creates LinkButton controls based on the value of its `dataProvider` property. Even though LinkBar is a subclass of Container, do not use methods such as `Container.addChild()` and `Container.removeChild()` to add or remove LinkButton controls. Instead, use methods such as `addItem()` and `removeItem()` to manipulate the `dataProvider` property. A LinkBar control automatically adds or removes the necessary children based on changes to the `dataProvider` property.

LinkButton control

The LinkButton control is part of the MX component set. There is no Spark equivalent.

The LinkButton control creates a single-line hypertext link that supports an optional icon. You can use a LinkButton control to open a URL in a web browser.

Creating a LinkButton control

You define a LinkButton control in MXML by using the `<mx:LinkButton>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block:

```
<?xml version="1.0"?>
<!-- controls\button\LBSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <mx:HBox>
        <mx:LinkButton label="link1"/>
        <mx:LinkButton label="link2"/>
        <mx:LinkButton label="link3"/>
    </mx:HBox>
</s:Application>
```

The following code contains a single LinkButton control that opens a URL in a web browser window:

```
<?xml version="1.0"?>
<!-- controls\button\LBURL.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <mx:LinkButton label="ADBE"
        width="100"
        click="navigateToURL(new URLRequest('http://quote.yahoo.com/q?s=ADBE'), 'quote')"/>
</s:Application>
```

This example uses the `navigateToURL()` method to open the URL.

The LinkButton control automatically provides visual cues when you move your mouse pointer over or click the control. The previous code example contains no link handling logic but does change color when you move your mouse pointer over or click a link.

The following code example contains LinkButton controls for navigating in a ViewStack navigator container:

```
<?xml version="1.0"?>
<!-- controls\button\LBViewStack.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <mx:VBox>
        <!-- Put the links in an HBox container across the top. -->
        <mx:HBox>
            <mx:LinkButton label="Link1"
                click="viewStack.selectedIndex=0;"/>
            <mx:LinkButton label="Link2"
                click="viewStack.selectedIndex=1;"/>
            <mx:LinkButton label="Link3"
                click="viewStack.selectedIndex=2;"/>
        </mx:HBox>
        <!-- This ViewStack container has three children. -->
        <mx:ViewStack id="viewStack">
            <mx:VBox width="150">
                <mx:Label text="One"/>
            </mx:VBox>
            <mx:VBox width="150">
                <mx:Label text="Two"/>
            </mx:VBox>
            <mx:VBox width="150">
                <mx:Label text="Three"/>
            </mx:VBox>
        </mx:ViewStack>
    </mx:VBox>
</s:Application>
```

A LinkButton control's label is centered within the bounds of the LinkButton control. You can position the text label in relation to the icon by using the `labelPlacement` property, which accepts the values `right`, `left`, `bottom`, and `top`.

LinkButton control user interaction

When a user clicks a LinkButton control, the LinkButton control dispatches a `click` event. If a LinkButton control is enabled, the following happens:

- When the user moves the mouse pointer over the LinkButton control, the LinkButton control changes its rollover appearance.
- When the user clicks the LinkButton control, the input focus moves to the control and the LinkButton control displays its pressed appearance. When the user releases the mouse button, the LinkButton control returns to its rollover appearance.
- If the user moves the mouse pointer off the LinkButton control while pressing the mouse button, the control's appearance returns to its original state and the control retains input focus.
- If the toggle property is set to `true`, the state of the LinkButton control does not change until the mouse button is released over the control.

If a LinkButton control is disabled, it appears as disabled, regardless of user interaction. In the disabled state, the control ignores all mouse or keyboard interaction.

NumericStepper control

The NumericStepper control is part of both the MX and Spark component sets. While you can use the MX NumericStepper control in your application, Adobe recommends that you use the Spark control instead.

You can use the NumericStepper control to select a number from an ordered set. The NumericStepper control consists of a single-line input text field and a pair of arrow buttons for stepping through the valid values. You can use the Up Arrow and Down arrow keys to cycle through the values.

If the user clicks the up arrow, the value displayed increases by one unit of change. If the user holds down the arrow, the value increases or decreases until the user releases the mouse button. When the user clicks the arrow, it is highlighted to provide feedback to the user.

Users can also type a legal value directly into the text field. Although editable ComboBox controls provide similar functionality, NumericStepper controls are sometimes preferred because they do not require a drop-down list that can obscure important data.

NumericStepper control arrows always appear to the right of the text field.

Creating a NumericStepper control

You define a NumericStepper control in MXML by using the `<s:NumericStepper>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block:

```
<?xml version="1.0"?>
<!-- controls\numericstepper\NumStepSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:NumericStepper id="nstepper1"
        value="6"
        stepSize="2"
        maximum="100"/>

</s:Application>
```

Sizing a NumericStepper control

When the control is resized, the width of the Up and Down arrow buttons in the NumericStepper control do not change size. If the NumericStepper control is sized greater than the default height, the associated stepper buttons appear pinned to the top and the bottom of the control.

User interaction

If the user clicks the Up or Down arrow button, the value displayed increases by one unit of change. If the user presses either of the arrow buttons for more than 200 milliseconds, the value in the input field increases or decreases, based on step size, until the user releases the mouse button or the maximum or minimum value is reached.

Keyboard navigation

The NumericStepper control has the following keyboard navigation features:

Key	Description
Down Arrow	Value decreases by one unit.
Up Arrow	Value increases by one unit.
Left Arrow	Moves the insertion point to the left within the NumericStepper control's text field.
Right Arrow	Moves the insertion point to the right within the Numeric Stepper control's text field.

To use the keyboard to navigate through the stepper, it must have focus.

PopUpAnchor control

The PopUpAnchor control is part of the Spark component set. There is no MX equivalent.

The PopUpAnchor control displays a pop-up component in a layout. It has no visual appearance, but it has dimensions. The PopUpAnchor control is used in the DropDownList control. The PopUpAnchor displays the pop-up component by adding it to the PopUpManager, and then sizes and positions the pop-up component relative to itself. Because the pop-up component is managed by the PopUpManager, it appears above all other controls.

With the PopUpAnchor control, you can create various kinds of popup functionality, such as the following:

- Click a button and a form to submit feedback pops up
- Click a button, and a search field pops up
- Mouse over the top of an application and a menu drops down
- In a calendar tool, double-click an appointment block to open an Edit dialog

Creating a pop-up component with PopUpAnchor

Define a PopUpAnchor control in MXML by using the `<s:PopUpAnchor>` tag. as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

Use the PopUpAnchor with two other components: the control that opens the popup, and the component that pops up (referred to as the pop-up component). The value of the `popUp` property is the pop-up component.

The following example creates an application with a button labeled Show slider. Click the button, and a VSlider component pops up. When you select a value using the slider, the `thumbRelease` event is triggered and the popup closes.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\popup\PopUpSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:HGroup>
        <s:Button label="Show slider"
            click="slide.displayPopUp = true"/>
        <s:PopUpAnchor id="slide">
            <s:VSlider
                maxHeight="40"
                thumbRelease="slide.displayPopUp = false"/>
        </s:PopUpAnchor>
    </s:HGroup>
</s:Application>
```

Sizing and positioning a popup component

The PopUpAnchor control sizes and positions the component that pops up (the pop-up component) relative to itself. If the pop-up doesn't fit, it will flip the position. ABOVE -> BELOW, RIGHT -> LEFT, etc. If it still doesn't fit, it will go back to the original position and call the pop-up until it is fully on screen.

The width of the pop-up component is determined in the following order:

- If `popUpWidthMatchesAnchorWidth` is true, use the actual width of the PopUpAnchor
- Use the pop-up component's `explicitWidth` value, if specified
- Use the pop-up component's `measuredWidth` value

The height of the pop-up component is determined in the following order:

- If `popUpHeightMatchesAnchorHeight` is true, use the actual height of the PopUpAnchor control
- Use the popup's `explicitHeight` value, if specified
- Use the pop-up component's `measuredHeight` value

The `popUpPosition` property controls the position of the pop-up component. Valid values are static properties of the PopUpPosition class and are as follows:

Value	Description
`above`	The bottom of the popup is adjacent to the top of the PopUpAnchor control. The left edge of the popup is vertically aligned with the left edge of the PopUpAnchor control.
`below`	The top of the popup is adjacent to the bottom of the PopUpAnchor control. The left edge of the pop-up is vertically aligned with the left edge of the PopUpAnchor control.
`left`	The right edge of the popup is adjacent to the left edge of the PopUpAnchor control. The top edge of the popup is horizontally aligned with the top edge of the PopUpAnchor control.
`right`	The left edge of the popup is adjacent to the right edge of the PopUpAnchor control. The top edge of the popup is horizontally aligned with the top edge of the PopUpAnchor control.
`center`	The horizontal and vertical center of the popup is positioned at the horizontal and vertical center of the PopUpAnchor control.
`topLeft`	The popup is positioned at the same default top-left position as the PopUpAnchor control.

Transforming and animating a popup component

Transformations include scaling, rotation, and skewing. Transformations that are applied to the PopUpAnchor control or its ancestors before the pop-up component opens are always applied to the pop-up component when it opens. However, if such changes are applied while the pop-up is open, the changes are not applied to the pop-up until the next time the pop-up opens. To apply transformations to the pop-up while it is open, call the `updatePopUpTransform()` method.

You can apply effects to animate the showing and hiding of the pop-up. Because transformations made on the PopUpAnchor control apply to its pop-up, you can apply effects to either the PopUpAnchor control or its pop-up component.

Consider the following when deciding whether to apply an effect on the PopUpAnchor control or its pop-up component:

- Apply Move, Fade, Resize, and Zoom effects to the PopUpAnchor control. Applying these effects to the PopUpAnchor control allows the effect to respect some of the PopUpAnchor control's layout constraints.
- Apply Mask and Shader effects on the pop-up component directly. These effects require applying a mask to the target or taking a bitmap snapshot of the target.

- If the effect is applied to the PopUpAnchor or its ancestors while the pop-up component is open, call `updatePopUpTransform()`. Calling this method reapplies the effect to the popup while the effect plays.

The following example uses states to control the display of the popup component. It uses transitions to play animations between states. When you click the Email button, an e-mail form fades into the application. Click Send or Cancel, and the e-mail form fades out. By only including the PopUpAnchor in the emailOpen state (`includeIn="emailOpen"`), the form is only instantiated when it is displayed.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >

    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>

    <fx:Style>
    .popUpBox
    {
        backgroundColor : #e9e9e9;
        borderStyle : "solid";
        paddingTop : 2;
        paddingBottom : 2;
        paddingLeft : 2;
        paddingRight : 2;
    }
    </fx:Style>
    <s:states>
        <s:State name="normal" />
        <s:State name="emailOpen" />
    </s:states>
    <s:transitions>
        <mx:Transition fromState="*" toState="*">
            <mx:Sequence>
                <s:Fade target="{emailPopUp.popUp}"
                    duration="250"/>
            </mx:Sequence>
        </mx:Transition>
    </s:transitions>
    <s:Group x="60">
        <s:Button label="Send email"
            click="currentState = 'emailOpen'"/>
        <s:PopUpAnchor id="emailPopUp"
            left="0" bottom="0"
            popUpPosition="below"
            styleName="popUpBox"
            includeIn="emailOpen"
            displayPopUp.normal="false"
            displayPopUp.emailOpen="true">
            <mx:Form>
                <mx:FormItem label="From :">
                    <s:TextInput/>
                </mx:FormItem>
                <mx:FormItem label="To :">
```

```
                    <s:TextInput/>
                </mx:FormItem>
                <mx:FormItem label="Subject :">
                    <s:TextInput/>
                </mx:FormItem>
                <mx:FormItem label="Message :">
                    <s:TextArea width="100%"
                        height="100"
                        maxChars="120"/>
                </mx:FormItem>
                <mx:FormItem direction="horizontal">
                    <s:Button label="Send"
                        click="currentState = 'normal'"/>
                    <s:Button label="Cancel"
                        click="currentState = 'normal'" />
                </mx:FormItem>
            </mx:Form>
        </s:PopUpAnchor>
    </s:Group>
</s:Application>
```

You can also control when the PopUpAnchor is destroyed. See “Create and apply view states” on page 1850, particularly the sections on “Controlling when to create added children” on page 1856 and “Controlling caching of objects created in a view state” on page 1858. These sections discuss the `itemCreationPolicy` and `itemDestructionPolicy` properties.

The following example displays a Search button with an animated popup. Click the button, and a text input field and a Find now button pops up on the right. Click the Find button, and the text input field and Find now button are hidden. The animation plays directly on the popup components.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Style>
        .popUpBox
        {
            backgroundColor : #e9e9e9;
            borderStyle : "solid";
            paddingTop : 2;
            paddingBottom : 2;
            paddingLeft : 2;
            paddingRight : 2;
        }
    </fx:Style>

    <fx:Declarations>
        <mx:Sequence id="hideSearch">
            <s:Scale target="{searchPopUp.popUp}"
                     scaleXFrom="1"
                     scaleXTo=".1"
                     duration="250"/>
            <mx:SetPropertyAction target="{searchPopUp}"
                                  name="displayPopUp"
                                  value="false"/>
        </mx:Sequence>
        <mx:Sequence id="showSearch">
            <mx:SetPropertyAction target="{searchPopUp}"
                                  name="displayPopUp"
                                  value="true"/>
            <s:Scale target="{searchPopUp.popUp}"
                     scaleXFrom=".1"
                     scaleXTo="1"
                     duration="200"/>
        </mx:Sequence>
    </fx:Declarations>

    <s:HGroup>
        <s:Button label="Search database" click="showSearch.play() "/>
        <s:PopUpAnchor id="searchPopUp"
                       left="0" right="0"
                       popUpPosition="right"
                       styleName="popUpBox">
            <s:HGroup>
                <s:TextInput id="searchField"
                             width="80" />
                <s:Button label="Find now"
                          click="hideSearch.play();"/>
            </s:HGroup>
        </s:PopUpAnchor>
    </s:HGroup>
</s:Application>
```

PopUpButton control

The PopUpButton control is part of the MX component set. There is no Spark equivalent.

The PopUpButton control consists of two horizontal buttons: a main button, and a smaller button called the pop-up button, which only has an icon. The main button is a Button control.

The pop-up button, when clicked, opens a second control called the pop-up control. Clicking anywhere outside the PopUpButton control, or in the pop-up control, closes the pop-up control

The PopUpButton control adds a flexible pop-up control interface to a Button control. One common use for the PopUpButton control is to have the pop-up button open a List control or a Menu control that changes the function and label of the main button.

The PopUpButton control is not limited to displaying menus; it can display any control as the pop-up control. A workflow application that lets users send a document for review, for example, could use a Tree control as a visual indication of departmental structure. The PopUpButton control's pop-up button would display the tree, from which the user could pick the message recipients.

The control that pops up does not have to affect the main button's appearance or action; it can have an independent action instead. You could create an undo PopUpButton control, for example, where the main button undoes only the last action, and the pop-up control is a List control that lets users undo multiple actions by selecting them.

The PopUpButton control is a subclass of the Button control and inherits all of its properties, styles, events, and methods, with the exception of the `toggle` property and the styles used for a selected button.

The control has the following characteristics:

- The `popUp` property specifies the pop-up control (for example, List or Menu).

- The `open()` and `close()` methods lets you open and close the pop-up control programmatically, rather than by using the pop-up button.
- The `open` and `close` events are dispatched when the pop-up control opens and closes.
- You use the `popUpSkin` and `arrowButtonWidth` style properties to define the PopUpButton control's appearance.

For detailed descriptions, see PopUpButton in *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a PopUpButton control

You use the `<mx:PopUpButton>` tag to define a PopUpButton control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

In the following example, you use the PopUpButton control to open a Menu control. Once opened, the Menu control, or any pop-up control, functions just as it would normally. You define an event listener for the Menu control's change event to recognize when the user selects a menu item, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\button\PopUpButtonMenu.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="600" width="600">
    <fx:Script>
        <![CDATA[
            import mx.controls.*;
            import mx.events.*;
            private var myMenu:Menu;
            // Initialize the Menu control,
            // and specify it as the pop up object
            // of the PopUpButton control.
            private function initMenu():void {
                myMenu = new Menu();
                var dp:Object = [
                  {label: "New Folder"},
                  {label: "Sent Items"},
                  {label: "Inbox"}
                ];
                myMenu.dataProvider = dp;
                myMenu.addEventListener("itemClick", changeHandler);
                popB.popUp = myMenu;
            }
            // Define the event listener for the Menu control's change event.
            private function changeHandler(event:MenuEvent):void {
                var label:String = event.label;
                popTypeB.text=String("Moved to " + label);
                popB.label = "Put in: " + label;
                popB.close();
            }
        ]]>
    </fx:Script>
    <mx:VBox>
        <mx:Label text="Main button mimics the last selected menuItem."/>
        <mx:PopUpButton id="popB"
            label="Edit"
            width="135"
            creationComplete="initMenu();"/>

        <mx:Spacer height="50"/>
        <mx:TextInput id="popTypeB"/>
    </mx:VBox>
</s:Application>
```

User interaction

You navigate the PopUpButton control in the following ways:

- Moving the mouse over any part of the PopUpButton control highlights the button border and the main button or the pop-up button.
- Clicking the button dispatches the `click` event.
- Clicking the pop-up button pops up the pop-up control and dispatches an `open` event.
- Clicking anywhere outside the PopUpButton control, or in the pop-up control, closes the pop-up control and dispatches a `close` event.

The following keystrokes let users navigate the PopUpButton control:

Key	Use
Spacebar	Behaves like clicking the main button.
Control+Down Arrow	Opens the pop-up control and initiates an `open` event. The pop-up control’s keyboard handling takes effect.
Control+Up Arrow	Closes the pop-up control and initiates a `close` event.

***Note:** You cannot use the Tab key to leave an opened pop-up control; you must make a selection or close the control with the Control+Up Arrow key combination.*

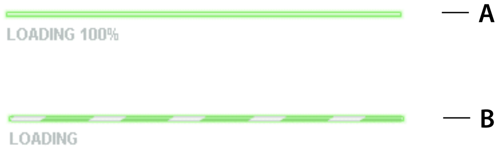
ProgressBar control

The ProgressBar control is part of the MX component set. There is no Spark equivalent.

The ProgressBar control provides a visual representation of the progress of a task over time. There are two types of ProgressBar controls: determinate and indeterminate. A *determinate* ProgressBar control is a linear representation of the progress of a task over time. You can use this when the user is required to wait for an extended period of time, and the scope of the task is known.

An *indeterminate* ProgressBar control represents time-based processes for which the scope is not yet known. As soon as you can determine the scope, you should use a determinate ProgressBar control.

The following example shows both types of ProgressBar controls:



Top. *Determinate ProgressBar control* ***Bottom.*** *Indeterminate ProgressBar control*

Use the ProgressBar control when the user is required to wait for completion of a process over an extended period of time. You can attach the ProgressBar control to any kind of loading content. A label can display the extent of loaded contents when enabled.

ProgressBar control modes

You use the `mode` property to specify the operating mode of the ProgressBar control. The ProgressBar control supports the following modes of operation:

event Use the `source` property to specify a loading process that emits `progress` and `complete` events. For example, the SWFLoader and Image controls emit these events as part of loading a file. You typically use a determinate ProgressBar in this mode. The ProgressBar control only updates if the value of the `source` property extends the EventDispatcher class. This is the default mode.

You also use this mode if you want to measure progress on multiple loads; for example, if you reload an image, or use the SWFLoader and Image controls to load multiple images.

polled Use the `source` property to specify a loading process that exposes the `bytesLoaded` and `bytesTotal` properties. For example, the SWFLoader and Image controls expose these properties. You typically use a determinate ProgressBar in this mode.

manual Set the `maximum`, `minimum`, and `indeterminate` properties along with calls to the `setProgress()` method. You typically use an indeterminate ProgressBar in this mode.

Creating a ProgressBar control

You use the `<mx:ProgressBar>` tag to define a ProgressBar control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example uses the default event mode to track the progress of loading an image by using the Image control:

```
<?xml version="1.0"?>
<!-- controls\pbar\PBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            public function initImage():void {
              image1.load('../assets/DSC00034.JPG');
            }
        ]]>
    </fx:Script>
    <mx:VBox id="vbox0"
        width="600" height="600">
        <mx:Canvas>
            <mx:ProgressBar width="200" source="image1"/>
        </mx:Canvas>
        <s:Button id="myButton"
            label="Show"
            click="initImage();"/>
        <mx:Image id="image1"
            height="500" width="600"
            autoLoad="false"
            visible="true"/>
    </mx:VBox>
</s:Application>
```

After you run this example the first time, the large image will typically be stored in your browser's cache. Before running this example a second time, clear your browser's cache so that you can see the ProgressBar control complete. If you do not clear your browser's cache, Flash Player loads the image from the cache and the ProgressBar control might go from 0% to 100% too quickly to see it tracking any progress in between.

In this mode, the Image control issues `progress` events during the load, and a `complete` event when the load completes.

The `<mx:Image>` tag exposes the `bytesLoaded` and `bytesTotal` properties, so you could also use `polled` mode, as the following example shows:

```
<mx:ProgressBar width="200" source="image1" mode="polled"/>
```

In manual mode, `mode="manual"`, you use an indeterminate ProgressBar control with the `maximum` and `minimum` properties and the `setProgress()` method. The `setProgress()` method has the following method signature:

```
setProgress(Number completed, Number total)
```

`completed` Specifies the progress made in the task, and must be between the `maximum` and `minimum` values. For example, if you were tracking the number of bytes to load, this would be the number of bytes already loaded.

`total` Specifies the total task. For example, if you were tracking bytes loaded, this would be the total number of bytes to load. Typically, this is the same value as `maximum`.

To measure progress, you make explicit calls to the `setProgress()` method to update the ProgressBar control.

To measure progress, you make explicit calls to the `setProgress()` method to update the ProgressBar control.

Defining the label of a ProgressBar control

By default, the ProgressBar displays the label *LOADING xx%*, where *xx* is the percent of the image loaded. You use the `label` property to specify a different text string to display.

The `label` property lets you include the following special characters in the label text string:

%1 Corresponds to the current number of bytes loaded.

%2 Corresponds to the total number of bytes.

%3 Corresponds to the percent loaded.

%% Corresponds to the % sign.

For example, to define a label that displays as:

```
Loading Image 1500 out of 78000 bytes, 2%
```

use the following code:

```
<?xml version="1.0"?>
<!-- controls\pbar\PBarLabel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            public function initImage():void {
              image1.load('../assets/DSC00034.JPG');
            }
        ]]>
    </fx:Script>
    <mx:VBox id="vbox0"
        width="600" height="600">
        <mx:Canvas>
            <mx:ProgressBar
                width="300"
                source="image1"
                mode="polled"
                label="Loading Image %1 out of %2 bytes, %3%%"
                labelWidth="400"
            />
        </mx:Canvas>
        <s:Button id="myButton"
            label="Show"
            click="initImage();"
        />
        <mx:Image id="image1"
            height="500" width="600"
            autoLoad="false"
            visible="true"
        />
    </mx:VBox>
</s:Application>
```

As with the previous example, be sure to clear your browser's cache before running this example a second time.

RadioButton control

The RadioButton and RadioButtonGroup controls are part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead.

The RadioButton control is a single choice in a set of mutually exclusive choices. A RadioButton group is composed of two or more RadioButton controls with the same group name. Only one member of the group can be selected at any given time. Selecting an deselected group member deselects the currently selected RadioButton control in the group.

While grouping RadioButton instances in a RadioButtonGroup is optional, a group lets you do things like set a single event handler on a group of buttons, rather than on each individual button. The RadioButtonGroup tag goes in the `<fx:Declarations>` tag.

Creating a RadioButton control

You define a RadioButton control in MXML by using the `<s:RadioButton>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\button\RBSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:VGroup>
        <s:RadioButton groupName="cardtype"
            id="americanExpress"
            label="American Express"
            width="150"/>
        <s:RadioButton groupName="cardtype"
            id="masterCard"
            label="MasterCard"
            width="150"/>
        <s:RadioButton groupName="cardtype"
            id="visa"
            label="Visa"
            width="150"/>
    </s:VGroup>
</s:Application>
```

For each RadioButton control in the group, you can optionally define an event listener for each button's `click` event. When a user selects a RadioButton control, Flex calls the event handler associated with that button's `click` event, as the following code example shows:

```
<?xml version="1.0"?>
<!-- controls\button\RBEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            private function handleAmEx(event:Event):void {
                // Handle event.
                myTA.text="Got Amex";
            }
            private function handleMC(event:Event):void {
                // Handle event.
                myTA.text="Got MasterCard";
            }
            private function handleVisa(event:Event):void {
                // Handle event.
                myTA.text="Got Visa";
            }
        ]]>
    </fx:Script>

    <s:VGroup>
        <s:RadioButton groupName="cardtype"
            id="americanExpress"
            label="American Express"
            width="150"
            click="handleAmEx(event);"/>
        <s:RadioButton groupName="cardtype"
            id="masterCard"
            label="MasterCard"
            width="150"
            click="handleMC(event);"/>
        <s:RadioButton groupName="cardtype"
            id="visa"
            label="Visa"
            width="150"
            click="handleVisa(event);"/>
        <s:TextArea id="myTA"/>
    </s:VGroup>
</s:Application>
```

Creating a group by using the <s:RadioButtonGroup> tag

The previous example created a RadioButton group by using the `groupName` property of each RadioButton control. To set functionality such as a single event handler on the group, use the `<s:RadioButtonGroup>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\button\RBGroupSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.events.ItemClickEvent;
            private function handleCard(event:ItemClickEvent):void {
                //Print the value of the selected RadioButton in the Text Area
                var cardValue:Object = cardtype.selectedValue;
                myTA.text="You selected " + cardValue;
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:RadioButtonGroup id="cardtype"
            itemClick="handleCard(event);"/>
    </fx:Declarations>

    <s:VGroup>
        <s:RadioButton group="{cardtype}"
            id="americanExpress"
            label="American Express"
            width="150"/>
        <s:RadioButton group="{cardtype}"
            id="masterCard"
            label="MasterCard"
            width="150"/>
        <s:RadioButton group="{cardtype}"
            id="visa"
            label="Visa"
            width="150"/>
        <s:TextArea id="myTA"/>
    </s:VGroup>
</s:Application>
```

In this example, the `id` property of the `<s:RadioButtonGroup>` tag defines the group name. use data binding to associate the group name with the `group` property of each RadioButton control.

The single `itemClick` event listener is defined for all buttons in the group. The `id` property is required when you use the `<s:RadioButtonGroup>` tag. The RadioButtonGroup is declared using the `<fx:Declarations>` tag.

The `itemClick` event listener for the group can take a different action depending on which button was selected, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\button\RBGroupEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.ItemClickEvent;

            private function handleCard(event:ItemClickEvent):void {
                if (event.currentTarget.selectedValue == "AmEx") {
                    Alert.show("You selected American Express.");
                } else if (event.currentTarget.selectedValue == "MC") {
                        Alert.show("You selected MasterCard.");
                } else {
                    Alert.show("You selected Visa.");
                }
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:RadioButtonGroup id="cardtype"
            itemClick="handleCard(event);"/>
    </fx:Declarations>

    <s:VGroup>
        <s:RadioButton group="{cardtype}"
            id="americanExpress"
            value="AmEx"
            label="American Express"
            width="150"/>
        <s:RadioButton group="{cardtype}"
            id="masterCard"
            value="MC"
            label="MasterCard"
            width="150"/>
        <s:RadioButton group="{cardtype}"
            id="visa"
            value="Visa"
            label="Visa"
            width="150"/>
    </s:VGroup>
</s:Application>
```

In the `itemClick` event listener, the `selectedValue` property of the RadioButtonGroup control in the event object is set to the value of the `value` property of the selected RadioButton control. If you omit the `value` property, Flex sets the `selectedValue` property to the value of the `label` property.

You can still define a `click` event listener for the individual buttons, even though you also define one for the group.

RadioButton user interaction

If a RadioButton control is enabled, when the user moves the pointer over an deselected RadioButton control, the button displays its rollover appearance. When the user clicks an deselected RadioButton control, the input focus moves to the control and the button displays its false pressed appearance. When the mouse button is released, the button displays the true state appearance. The previously selected RadioButton control in the group returns to its false state appearance.

If the user moves the pointer off the RadioButton control while pressing the mouse button, the control's appearance returns to the false pressed state. The control retains input focus.

If a RadioButton control is not enabled, the RadioButton control and RadioButton group display the disabled appearance, regardless of user interaction. In the disabled state, all mouse or keyboard interaction is ignored.

The RadioButton and RadioButtonGroup controls have the following keyboard navigation features:

Key	Action
Control+arrow keys	Move focus among the buttons without selecting a button.
Spacebar	Select a button.

HScrollBar and VScrollBar controls

The HScrollBar and VScrollBar controls are part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead.

The VScrollBar (vertical ScrollBar) control and HScrollBar (horizontal ScrollBar) controls let the user control the portion of data that is displayed when there is too much data to fit in the display area.

Although you can use the VScrollBar control and HScrollBar control as stand-alone controls, they are usually combined with other components as part of a custom component to provide scrolling functionality.

Scrollbar controls consists of four parts: two arrow buttons, a track, and a thumb. The position of the thumb and display of the buttons depends on the current state of the control. The width of the control is equal to the largest width of its subcomponents (arrow buttons, track, and thumb). Every subcomponent is centered in the scroll bar.

The ScrollBarBase control uses four parameters to calculate its display state:

- Minimum range value
- Maximum range value
- Current position value; must be within the minimum and maximum range values
- Viewport size; represents the number of items in the range that can be displayed at once and must be equal to or less than the range

For more information on using these controls with Spark containers, see "Scrolling Spark containers" on page 338.

Creating a scrollbar control

Define a scrollbar control in MXML by using the `<s:VScrollBar>` tag for a vertical scrollbar or the `<s:HScrollBar>` tag for a horizontal scrollbar, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\bar\SBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
     <fx:Script>
        <![CDATA[

            import mx.events.ScrollEvent;

            // Event handler function to display the scroll location.
            private function myScroll():void {
                showPosition.text = "VScrollBar properties summary:" + '\n' +
                    "------------------------------------" + '\n' +
                    "Current scroll position: " +
                    bar.value  + '\n' +
                    "The maximum scroll position: " +
                    bar.maximum + '\n' +
                    "The minimum scroll position: " +
                    bar.minimum;
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label
            width="100%"
            text="Click on the ScrollBar control to view its properties."/>

    <s:VScrollBar id="bar"
                   height="100%"
                   minimum="0"
                   maximum="{this.width - 20}"
                   stepSize="50"
                   pageSize="100"
                   repeatDelay="1000"
                   repeatInterval="500"
                   change="myScroll();"/>

    <s:TextArea id="showPosition"
                 height="100%"
                 width="100%" />
    </s:VGroup>
</s:Application>
```

Sizing a scrollbar control

The scrollbar control does not display correctly if it is sized smaller than the height of the Up arrow and Down arrow buttons. There is no error checking for this condition. Adobe recommends that you hide the scrollbar control in such a condition. If there is not enough room for the thumb, the thumb is made invisible.

User interaction

Use the mouse to click the various portions of the scrollbar control, which dispatches events to listeners. The object listening to the scrollbar control is responsible for updating the portion of data displayed. The scrollbar control updates itself to represent the new state after the action has taken place.

Scroller control

The Scroller control is part of the Spark component set. There is no MX equivalent.

The Scroller control contains a pair of scroll bars and a viewport. A viewport displays a rectangular subset of the area of a component, rather than displaying the entire component. You can use the Scroller control to make any container that implements the IViewport interface, such as Group, scrollable.

The scroll bars control the viewport's vertical and horizontal scroll position. They reflect the viewport's actual size and content size. Scroll bars are displayed according to the Scroller's vertical and horizontal scroll policy properties. By default, the policy is set to `"auto"`. This value indicates that scroll bars are displayed when the content within a viewport is larger than the actual size of the viewport.

For more information on using this control with Spark containers, see “Scrolling Spark containers” on page 338.

Creating a Scroller control

You define a Scroller control in MXML by using the `<s:Scroller>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

In the following example, the Group container `myGroup` is the viewport for this scroller. The content in the viewport is the loaded image. The layout of the application is controlled by the VGroup container.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\Scroller\ScrollerImage.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="450" height="450">
    <s:VGroup paddingLeft="10" paddingTop="10">
        <s:Scroller width="400" height="400">
            <s:Group id="myGroup" >
                <s:Image id="loader1"
                          source="@Embed(source='../assets/strawberries.jpg')"/>
            </s:Group>
        </s:Scroller>
        <s:Label
            fontSize="14" fontFamily="Arial"
            text= "Scroll to see fresh summer strawberries"/>
    </s:VGroup>
</s:Application>
```

Sizing a Scroller control

You can size the width and height of the Scroller control directly or size the viewport container. The VScrollBar and HScrollBar classes bind to the viewport's scroll position and actual and content sizes.

Skinning a Scroller control

The Scroller skin provides scroll bars and manages layout according to the verticalScrollPolicy and horizontalScrollPolicy properties in the Scroller class.

The Scroller skin layout cannot be changed because it must handle the vertical and horizontal scroll policies. Scroller skins can only provide replacement scroll bars.

Spinner control

The Spinner control is part of the Spark component set. There is no MX equivalent.

The Spinner control lets users step through an allowed set of values and select a value by clicking up or down buttons. It is a base class for the NumericStepper control. You could use Spinner to create controls with a different input or display than the standard text input field used in NumericStepper. Another possibility is to use a Spinner control to control tabs or a menu by assigning different values to tabs or menu items.

Creating a Spinner control

You define a Spinner control in MXML by using the `<s:Spinner>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example shows a Spinner control that allows wrapping back to the minimum value once the maximum value is reached. The `allowValueWrap` property is false by default. The `snapInterval` property is set to 2. When you click the up (or increment) button for the first time, the value of the Spinner control is set to 4.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- \controls\spinner\SpinnerSimple.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            private function onClickHandler(event:Event):void {
                message.text += "You selected "+ mySpinner.value + "\n";
                }
        ]]>
    </fx:Script>

    <s:VGroup paddingTop="10" paddingLeft="10">
        <s:HGroup>
            <s:Label text="Use the arrows to change value"/>
            <s:Spinner id="mySpinner"
                       click="onClickHandler(event);"
                       minimum="2"
                       maximum="20"
                       snapInterval="2"
                       allowValueWrap="true"/>
        </s:HGroup>
        <s:TextArea id="message"/>
    </s:VGroup>
</s:Application>
```

SWFLoader control

The SWFLoader control is part of the MX component set. There is no Spark equivalent of this control.

The SWFLoader control lets you load one Flex application into another Flex application as a SWF file. It has properties that let you scale its contents. It can also resize itself to fit the size of its contents. By default, content is scaled to fit the size of the SWFLoader control. The SWFLoader control can also load content on demand programmatically, and monitor the progress of a load operation.

When loading an applications into a main application, you should be aware of the following factors:

Versioning SWF files produced with earlier versions of Flex or ActionScript may not work properly when loaded with the SWFLoader control. You can use the `loadForCompatibility` property of the SWFLoader control to ensure that applications loaded into a main application works, even if the applications were compiled with a different version of the compiler.

Security When loading applications, especially ones that were created by a third-party, you should consider loading them into their own SecurityDomain. While this places additional limitations on the level of interoperability between the main application and the application, it ensures that the content is safe from attack.

For more information about creating and loading applications, see “Developing and loading sub-applications” on page 176.

The SWFLoader control also lets you load the contents of a GIF, JPEG, PNG, SVG, or SWF file into your application, where the SWF file does not contain a Flex application, or a ByteArray representing a SWF, GIF, JPEG, or PNG.

For more information, see “Image control” on page 676. For more information on using the SWFLoader control to load a Flex application, see “Externalizing application classes” on page 719.

***Note:** Flex also includes the Image control for loading GIF, JPEG, PNG, SVG, or SWF files. You typically use the Image control for loading static graphic files and SWF files, and use the SWFLoader control for loading Flex applications as SWF files. The Image control is also designed to be used in custom cell renderers and item editors.*

A SWFLoader control cannot receive focus. However, content loaded into the SWFLoader control can accept focus and have its own focus interactions.

Creating a SWFLoader control

You define a SWFLoader control in MXML by using the `<mx:SWFLoader>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block:

```
<?xml version="1.0"?>
<!-- controls\swfloader\SWFLoaderSimple.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:VGroup>
        <mx:SWFLoader id="loader1" source="FlexApp.swf"/>
    </s:VGroup>

</s:Application>
```

Like the Image control, you can also use the Embed statement with the SWFLoader control to embed the image in your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\swfloader\SWFLoaderSimpleEmbed.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:SWFLoader id="loader1" source="@Embed(source='FlexApp.swf')"/>

</s:Application>
```

When using the SWFLoader control with an SVG file, you can only load it by using an Embed statement; you cannot load an SVG file at run time. For more information about embedding resources, see the description for the Image control at “Image control” on page 676, and “Embedding assets” on page 1699.

This technique works well with SWF files that add graphics or animations to an application but are not intended to have a large amount of user interaction. If you import SWF files that require a large amount of user interaction, you should build them as custom components.

Interacting with a loaded Flex application

The following example, in the file FlexApp.mxml, shows a simple Flex application that defines two Label controls, a variable, and a method to modify the variable:

```
<?xml version="1.0"?>
<!-- controls\swfloader\FlexApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="100" width="200">
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var varOne:String = "This is a public variable.";
            public function setVarOne(newText:String):void {
                varOne=newText;
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label id="lblOne" text="I am here."/>
        <s:Label text="{varOne}"/>
        <s:Button label="Nested Button" click="setVarOne('Nested button pressed.');"/>
    </s:VGroup>
</s:Application>
```

You compile this example into the file FlexApp.SWF, and then use the SWFLoader control to load it into another Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\swfloader\SWFLoaderInteract.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    backgroundColor="gray"
    height="200">
    <fx:Script>
        <![CDATA[
            import mx.managers.SystemManager;

            [Bindable]
            public var loadedSM:SystemManager;
            // Initialize variables with information from
            // the loaded application.
            private function initNestedAppProps():void {
                loadedSM = SystemManager(myLoader.content);

                // Enable the buttons after the application loads.
                b1.enabled = true;
                b2.enabled = true;
                b3.enabled = true;
            }
            // Update the Label control in the outer application
            // from the Label control in the loaded application.
            public function updateLabel():void {
                lbl.text=loadedSM.application["lblOne"].text;
            }
            // Write to the Label control in the loaded application.
            public function updateNestedLabels():void {
                loadedSM.application["lblOne"].text = "I was just updated.";
                loadedSM.application["varOne"] = "I was just updated.";
            }

            // Write to the varOne variable in the loaded application
            // using the setVarOne() method of the loaded application.
            public function updateNestedVarOne():void {
                loadedSM.application["setVarOne"]("Updated varOne!");
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label id="lbl"/>
        <mx:SWFLoader id="myLoader" width="300"
            source="FlexApp.swf"
            complete="initNestedAppProps();"/>
        <s:Button id="b1" label="Update Label Control in Outer Application"
            click="updateLabel();"
            enabled="false"/>
        <s:Button id="b2" label="Update Nested Controls"
            click="updateNestedLabels();"
            enabled="false"/>
        <s:Button id="b3" label="Update Nested varOne"
            click="updateNestedVarOne();"
            enabled="false"/>
    </s:VGroup>
</s:Application>
```

In this example, the FlexApp.swf file is in the same directory as the main application. Modify the path to the file appropriately for your application.

Notice that this application loads the SWF file at run time; it does not embed it. For information on embedding a Flex application by using the SWFLoader tag, see “Embedding assets” on page 1699.

You use the `complete` event of the SWFLoader control to initialize a variable with a reference to the SystemManager object for the loaded Flex application.

When a user clicks the first Button control in the outer application, Flex copies the text from the Label control in the loaded application to the Label control in the outer application.

When a user clicks the second Button control, Flex writes the text to the Label control and to the *varOne* variable defined in the loaded application.

When a user clicks the third Button control, the *setVarOne()* method of the loaded application writes to the *varOne* variable defined in the loaded application.

Note: *When working with loaded SWF files, consider using modules rather than the SWFLoader control. For more information on creating modular applications, see “Modular applications” on page 138.*

The level of interoperability between a main application and a loaded application depends on the type of loaded application:

Sandboxed Sandboxed applications are loaded into their own security domains, and can be multi-versioned. Using sandboxed applications is the recommended practice for third-party applications. In addition, if your loaded applications use RPC or DataServices-related functionality, you should load them as sandboxed. For more information, see “Developing sandboxed applications” on page 205.

Multi-versioned Multi-versioned applications are those that can be compiled with different versions of the Flex framework than the main application that loads them. Their interoperability with the main application and other loaded applications is more limited than single-versioned applications. For more information, see "Developing multi-versioned applications" on page 213.

Single-versioned Single-versioned applications are applications that are guaranteed to have been compiled with the same version of the compiler as the main application. They have the greatest level of interoperability with the main application, but they also require that you have complete control over the source of the loaded applications. For more information, see "Creating and loading sub-applications" on page 191.

Externalizing application classes

To reduce the size of the applications that you load by using the SWFLoader control, you can instruct the loaded application to externalize framework classes that are also included by the loading application. The result is that the loaded application is smaller because it only includes the classes it requires, while the framework code and other dependencies are included in the loading application.

To externalize framework classes, you generate a linker report from the loading application by using `link-report` option to the mxmlc command. You then use the `load-externs` option to the mxmlc compiler to specify this report when you compile the loaded application.

Externalize framework classes

1 Generate the linker report for the loading application:

```
mxmlc -link-report=report.xml MyApplication.mxml
```

2 Compile the loaded application by using the link report:

```
mxmlc -load-externs=report.xml MyLoadedApplication.mxml
```

3 Compile the loading application:

```
mxmlc MyApplication.mxml
```

***Note:** If you externalize the loaded application's dependencies by using the `load-externs` option, your loaded application might not be compatible with future versions of Adobe Flex. Therefore, you might be required to recompile the application. To ensure that a future Flex application can load your application, compile that module with all the classes it requires.*

For more information, see "Flex compilers" on page 2164.

Sizing a SWFLoader control

You use the SWFLoader control's `scaleContent` property to control the sizing behavior of the SWFLoader control. When the `scaleContent` property is set to `true`, Flex scales the content to fit within the bounds of the control. However, images will still retain their aspect ratio by default.

Styling loaded applications

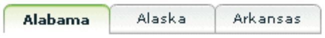
When you use the SWFLoader control to load a loaded application, the loaded application uses the same rules for applying styles as modules.

For more information, see "Using styles with modules" on page 143.

MX TabBar control

The TabBar control is part of both the MX and Spark component sets. While you can still use the MX control in your application, Adobe recommends that you use the Spark control instead. For information on Spark TabBar, see "Spark ButtonBar and TabBar controls" on page 529.

A TabBar control defines a horizontal or vertical row of tabs. The following shows an example of a TabBar control:



As with the LinkBar control, you can use a TabBar control to control the active child container of a ViewStack container. The syntax for using a TabBar control to control the active child of a ViewStack container is the same as for a LinkBar control. For an example, see "MX ViewStack navigator container" on page 630.

While a TabBar control is similar to a TabNavigator container, it does not have any children. For example, you use the tabs of a TabNavigator container to select its visible child container. You can use a TabBar control to set the visible contents of a single container to make that container's children visible or invisible based on the selected tab.

Creating an MX TabBar control

You use the `<mx:TabBar>` tag to define a TabBar control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the data for the TabBar control by using the `<mx:dataProvider>` child tag of the `<mx:TabBar>` tag. The `<mx:dataProvider>` tag lets you specify data in several different ways. In the simplest case for creating a TabBar control, you use the `<mx:dataProvider>` and `<fx:String>` tags to specify the text for each tab, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:TabBar>
        <mx:dataProvider>
            <fx:String>Alabama</fx:String>
            <fx:String>Alaska</fx:String>
            <fx:String>Arkansas</fx:String>
        </mx:dataProvider>
    </mx:TabBar>
</s:Application>
```

The `<fx:String>` tags define the text for each tab in the TabBar control.

You can also use the `<fx:Object>` tag to define the entries as an array of objects, where each object contains a `label` property and an associated data value, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarObject.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:TabBar>
        <mx:dataProvider>
            <fx:Object label="Alabama" data="Montgomery"/>
            <fx:Object label="Alaska" data="Juneau"/>
            <fx:Object label="Arkansas" data="Little Rock"/>
        </mx:dataProvider>
    </mx:TabBar>
</s:Application>
```

The `label` property contains the state name and the `data` property contains the name of its capital. The `data` property lets you associate a data value with the text label. For example, the `label` text could be the name of a color, and the associated data value could be the numeric representation of that color.

By default, Flex uses the value of the `label` property to define the tab text. If the object does not contain a `label` property, you can use the `labelField` property of the TabBar control to specify the property name containing the tab text, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarLabel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:TabBar labelField="state">
        <mx:dataProvider>
            <fx:Object state="Alabama" data="Montgomery"/>
            <fx:Object state="Alaska" data="Juneau"/>
            <fx:Object state="Arkansas" data="Little Rock"/>
        </mx:dataProvider>
    </mx:TabBar>
</s:Application>
```

Passing data to an MX TabBar control

Flex lets you populate a TabBar control from an ActionScript variable definition or from a Flex data model. When you use a variable, you can define it to contain one of the following:

- A label (string)
- A label (string) paired with data (scalar value or object)

The following example populates a TabBar control from a variable:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarVar.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var STATE_ARRAY:ArrayCollection = new ArrayCollection([
                {label:"Alabama", data:"Montgomery"},
                {label:"Alaska", data:"Juneau"},
                {label:"Arkansas", data:"LittleRock"}
            ]);
        ]]>
    </fx:Script>
    <mx:TabBar >
        <mx:dataProvider>
            {STATE_ARRAY}
        </mx:dataProvider>
    </mx:TabBar>
</s:Application>
```

You can also bind a Flex data model to the `dataProvider` property. For more information on using data models, see "Storing data" on page 889.

Handling MX TabBar control events

The TabBar control defines an item`Click` event that is broadcast when a user selects a tab. The event object contains the following properties:

- `label` String containing the label of the selected tab.
- `index` Number containing the index of the selected tab. Indexes are numbered from 0 to *n* - 1, where *n* is the total number of tabs. The default value is 0, corresponding to the first tab.

The following example code shows a handler for the `itemClick` event for this TabBar control:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.events.ItemClickEvent;
            import mx.controls.TabBar;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var STATE_ARRAY:ArrayCollection = new ArrayCollection([
                {label:"Alabama", data:"Montgomery"},
                {label:"Alaska", data:"Juneau"},
                {label:"Arkansas", data:"LittleRock"}
            ]);
            private function clickEvt(event:ItemClickEvent):void {
                // Access target TabBar control.
                var targetComp:TabBar = TabBar(event.currentTarget);
                forClick.text="label is: " + event.label + "\nindex is: " +
                    event.index + "\ncapital is: " +
                    targetComp.dataProvider[event.index].data;
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <mx:TabBar id="myTB" itemClick="clickEvt(event);">
            <mx:dataProvider>
                {STATE_ARRAY}
            </mx:dataProvider>
        </mx:TabBar>

        <s:TextArea id="forClick" width="250" height="100"/>
    </s:VGroup>
</s:Application>
```

In this example, every `itemClick` event updates the TextArea control with the tab label, selected index, and the selected data from the TabBar control's `dataProvider` Array.

MX VideoDisplay control

The VideoDisplay control is part of both the MX and Spark component sets. While you can still use the MX control in your application, Adobe recommends that you use the Spark control instead. Continue to use the MX VideoDisplay control to work with cue points or stream live video from a local camera using the `attachCamera()` method of the Camera class.

Flex supports the VideoDisplay control to incorporate streaming media into Flex applications. Flex supports the Flash Video File (FLV) file format with this control.

Using media in Flex

Media, such as movie and audio clips, are used more and more to provide information to web users. As a result, you need to provide users with a way to stream the media, and then control it. The following examples are usage scenarios for media controls:

- Showing a video message from the CEO of your company
- Streaming movies or movie previews
- Streaming songs or song snippets
- Providing learning material in the form of media

The streaming VideoDisplay control makes it easy to incorporate streaming media into Flash presentations. Flex supports the Flash Video File (FLV) file format with this control. You can use this control with video and audio data. When you use the VideoDisplay control by itself, your application provides no mechanism for its users to control the media files.

***Note:** The VideoDisplay control does not support scan forward and scan backward functionality. Also, the VideoDisplay control does not support accessibility or styles.*

About the MX VideoDisplay control

Flex creates an MX VideoDisplay control with no visible user interface. It is simply a control to hold and play media.

***Note:** The user cannot see anything unless some video media is playing.*

The `playheadTime` property of the control holds the current position of the playhead in the video file, measured in seconds. Most events dispatched by the control include the playhead position in the associated event object. One use of the playhead position is to dispatch an event when the video file reaches a specific position. For more information, see “Adding a cue point to the MX VideoDisplay control” on page 725.

The VideoDisplay control also supports the `volume` property. This property takes a value from 0.0 to 1.00; 0.0 is mute and 1.00 is the maximum volume. The default value is 0.75.

Setting the size of the MX VideoDisplay control

The appearance of any video media playing in a VideoDisplay control is affected by the following properties:

- `maintainAspectRatio`
- `height`
- `width`

When you set `maintainAspectRatio` to `true` (the default), the control adjusts the size of the playing media after the control size has been set. The size of the media is set to maintain its aspect ratio.

If you omit both `width` and `height` properties for the control, Flex makes the control the size of the playing media. If you specify only one property, and the `maintainAspectRatio` property is `false`, the size of the playing media determines the value of the other property. If the `maintainAspectRatio` property is `true`, the media retains its aspect ratio when resizing.

The following example creates a VideoDisplay control:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplaySimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:VBox>
        <mx:VideoDisplay
            source="../assets/MyVideo.flv"
            height="250"
            width="250"
        />
    </mx:VBox>
</s:Application>
```

By default, Flex sizes the VideoDisplay control to the size of the media. If you specify the `width` or `height` property of the control, and either is smaller than the media's dimensions, Flex does not change the size of the component. Instead, Flex sizes the media to fit within the component. If the control's playback area is smaller than the default size of the media, Flex shrinks the media to fit inside the control.

Using methods of the MX VideoDisplay control

You can use the following methods of the VideoDisplay control in your application: `close()`, `load()`, `pause()`, `play()`, and `stop()`. The following example uses the `pause()` and `play()` methods in event listener for two Button controls to pause or play an FLV file:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayStopPlay.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="myVid.pause();">
    <mx:VBox>
        <mx:VideoDisplay id="myVid"
            source="../assets/MyVideo.flv"
            height="250"
            width="250"
        />
        <mx:HBox>
            <mx:Button label="||" click="myVid.pause();"/>
            <mx:Button label="&gt;" click="myVid.play();"/>
        </mx:HBox>
    </mx:VBox>
</s:Application>
```

Adding a cue point to the MX VideoDisplay control

You can use cue points to trigger events when the playback of your media reaches a specified location. To use cue points, you set the `cuePointManagerClass` property to mx.controls.videoClasses.CuePointManager to enable cue point management, and then pass an Array of cue points to the `cuePoints` property of the VideoDisplay control. Each element of the Array contains two fields. The `name` field contains an arbitrary name of the cue point. The `time` field contains the playhead location, in seconds, of the VideoDisplay control with which the cue point is associated.

When the playhead of the VideoDisplay control reaches a cue point, it dispatches a `cuePoint` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayCP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="450"
    creationComplete="myVid.pause();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.CuePointEvent;
            import mx.controls.videoClasses.CuePointManager;
            private function cpHandler(event:CuePointEvent):void {
                cp.text+="Got to " + event.cuePointName + " cuepoint @ " +
                    String(event.cuePointTime) + " seconds in.\n";
            }
        ]]>
    </fx:Script>
    <mx:VBox>
        <mx:VideoDisplay id="myVid"
            source="../assets/MyVideo.flv"
            cuePointManagerClass="mx.controls.videoClasses.CuePointManager"
            cuePoint="cpHandler(event);"
            height="250"
            width="250"
        >
            <mx:cuePoints>
                <fx:Object name="first" time="5"/>
                <fx:Object name="second" time="10"/>
            </mx:cuePoints>
        </mx:VideoDisplay>
        <mx:Label text="{myVid.playheadTime}"/>
        <mx:TextArea id="cp" height="100" width="250"/>
    </mx:VBox>
    <mx:HBox>
        <mx:Button label="||" click="myVid.pause();"/>
        <mx:Button label="&gt;" click="myVid.play();"/>
    </mx:HBox>
</s:Application>
```

In this example, the event listener writes a String to the TextArea control when the control reaches a cue point. The String contains the name and time of the cue point.

Adding a cue point by using the CuePointManager class

You can set cue points for the VideoDisplay control by using the `cuePointManager` property. This property is of type CuePointManager, where the CuePointManager class defines methods that you use to programmatically manipulate cue points, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayCPManager.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="450"
    creationComplete="myVid.pause();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.events.CuePointEvent;
            [Bindable]
            private var myCuePoints:Array = [
                { name: "first", time: 5},
                { name: "second", time: 10}
            ];

            // Set cue points using methods of the CuePointManager class.
            private function initCP():void {
                myVid.cuePointManager.setCuePoints(myCuePoints);
            }
            private var currentCP:Object=new Object();
            private function cpHandler(event:CuePointEvent):void {
                cp.text += "Got to " + event.cuePointName + " cuepoint @ " +
                    String(event.cuePointTime) + " seconds in.\n";
                // Remove cue point.
                currentCP.name=event.cuePointName;
                currentCP.time=event.cuePointTime;
                myVid.cuePointManager.removeCuePoint(currentCP);
                // Display the number of remaining cue points.
                cp.text += "# cue points remaining: " +
                    String(myVid.cuePointManager.getCuePoints().length) + ".\n";
            }
        ]]>
    </fx:Script>
    <mx:VBox>
        <mx:VideoDisplay id="myVid"
            cuePointManagerClass="mx.controls.videoClasses.CuePointManager"
            source="../assets/MyVideo.flv"
            cuePoint="cpHandler(event);"
            height="250"
            width="250"
        />
        <mx:Label text="{myVid.playheadTime}"/>
        <mx:TextArea id="cp" height="100" width="250"/>
    </mx:VBox>
    <mx:HBox>
        <mx:Button label="&gt;" click="cp.text='';initCP();myVid.play();"/>
    </mx:HBox>
</s:Application>
```

Streaming video from a camera to the MX VideoDisplay control

You can use the `VideoDisplay.attachCamera()` method to configure the control to display a video stream from a camera, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayCamera.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[

            // Define a variable of type Camera.
            import flash.media.Camera;
            public var cam:Camera;
            public function initCamera():void {
                // Initialize the variable.
                cam = Camera.getCamera();
                myVid.attachCamera(cam)
            }
        ]]>
    </fx:Script>

    <mx:VideoDisplay id="myVid"
        width="320" height="240"
        creationComplete="initCamera();"/>
</s:Application>
```

In this example, you create a Camera object in the event handler for the `creationComplete` event of the VideoDisplay control, then pass the Camera object as the argument to the `attachCamera()` method.

Using the MX VideoDisplay control with Flash Media Server

You can use the VideoDisplay control to play a media stream from Adobe® Flash® Media Server, without needing to register the Flex application with the server. The following code shows a simple example that uses the video-on-demand (vod) service that is available in Flash Media Server 3.5 and later:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayFMS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>

    <s:Label text="RTMP FMS 3.5"/>
    <mx:VideoDisplay
            source="rtmp://fmsexamples.adobe.com/vod/edge_codeJam.flv"/>
</s:Application>
```

Spark VideoPlayer and VideoDisplay controls

The VideoPlayer control lets you play progressively downloaded or streaming video. It supports multi-bit rate streaming and live video when used with a server that supports these features, such as Flash Media Server 3.5 or later.

The VideoPlayer control contains a full UI to let users control playback of video. It contains a play/pause toggle button; a scrub bar to let users seek through video; a volume bar; a timer; and a button to toggle in and out of fullscreen mode.

Flex also offers the Spark VideoDisplay control, which plays video without any chrome, skin, or UI. The Spark VideoDisplay has the same methods and properties as the Spark VideoPlayer control. It is useful when you do not want the user to interact with the control, or when you want to fully customize the interactivity but not reskin the VideoPlayer control.

Both VideoPlayer and VideoDisplay support playback of FLV and F4V file formats, as well as MP4-based container formats.

About OSMF

The underlying implementation of the Spark VideoDisplay and VideoPlayer classes rely on the Open Source Media Framework (OSMF) classes.

You can use the OSMF classes to extend the VideoPlayer and VideoDisplay classes. For example, you can use OSMF to incorporate advertising, reporting, cue points, and DVR functionality.

You can also use OSMF classes to implement your own player. For an example of this, see OSMF + Flex Example.

For more information about OSMF, see Open Source Media Framework

Spark VideoPlayer events

The VideoPlayer control dispatches several different types of events that you can use to monitor and control playback. The event objects associated with each event dispatched by the VideoPlayer control is defined by a class in the org.osmf.events.* package.

The following example handles the `complete` and `mediaPlayerStateChange` events dispatched by the VideoPlayer control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\videoplayer\VideoPlayerEvent.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import org.osmf.events.MediaPlayerStateChangeEvent;
            import org.osmf.events.TimeEvent;
            protected function vpCompleteHandler(event:TimeEvent):void {
                myTA.text = "Video complete - restarting."
            }
            protected function
vpMediaPlayerStateChangeHandler(event:MediaPlayerStateChangeEvent):void {
                if (event.state == "loading")
                    myTA.text = "loading ...";
                if (event.state == "playing")
                    myTA.text = "playing ...";
            }
        ]]>
    </fx:Script>

    <s:VideoPlayer
        source="rtmp://fmsexamples.adobe.com/vod/mp4:_cs4promo_1000.f4v"
        width="350" height="250"
        loop="true"
        complete="vpCompleteHandler(event);"
        mediaPlayerStateChange="vpMediaPlayerStateChangeHandler(event);"/>
    <s:TextArea id="myTA" width="350" height="25"/>
</s:Application>
```

The VideoPlayer control dispatches the `complete` event when the video completes. The event object for the complete event is of type org.osmf.events.TimeEvent.

The VideoPlayer control dispatches the `mediaPlayerStateChange` event when the state of the control changes. The control has several states, including the `buffering`, `loading`, `playing`, and `ready` states. The event object for the `mediaPlayerStateChange` event is of org.osmf.events.MediaPlayerStateChangeEvent. The event handler for the `mediaPlayerStateChange` event uses the `state` property of the event object to determine the new state of the control.

Setting the video source for the Spark VideoPlayer control

The VideoPlayer component supports playback of local media, streaming media, and progressively downloaded media. You can play back both live and recorded media.

To play back a single media file, use the `source` attribute. The correct value for the `source` attribute is the path to the file. Use the correct syntax in the URL: HTTP for progressive download and RTMP for streaming from Flash Media Server. If using Flash Media Server, use the correct prefixes or extensions.

The following code plays a local FLV file, phone.flv, that resides in the assets folder in the same project folder as the SWF file:

```
<s:VideoPlayer source="assets/phone.flv"/>
```

The following code plays a single F4V file from Flash Media Server, using the video-on-demand service that Flash Media Server 3.5 and later provides. The use of the MP4 prefix and the F4V file extension are required. The MP4 prefix is always required; the F4V file extension is required when the name of the file to be loaded contains a file extension.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\videoplayer\VideoPlayerSimple.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:VideoPlayer
        source="rtmp://fmsexamples.adobe.com/vod/mp4:_cs4promo_1000.f4v"
        width="350" height="250"
        loop="true"/>
</s:Application>
```

Using dynamic streaming

The Spark VideoPlayer and VideoDisplay components support dynamic streaming (also known as *multi-bitrate streaming* or *adaptive streaming*) for on-demand and live content. To use dynamic streaming, you encode a stream at several different bit rates. During playback, Flash Player dynamically switches among the streams based on connection quality and other metrics to provide the best experience.

You can do dynamic streaming over RTMP and HTTP. On-demand and live dynamic streaming over RTMP require Flash Media Server. RTMP streaming has the following benefits over HTTP streaming:

- RTMPe for lightweight content protection
- RTMP quality of service
- Absolute timecodes
- Enhanced buffering

Live HTTP Dynamic Streaming requires Flash Media Server. On-demand HTTP Dynamic Streaming requires an Apache HTTP Server Origin Module and a File Packager tool. These tools are available as free downloads from http://www.adobe.com/products/httpdynamicstreaming/.

Dynamic streaming over HTTP

To use live and on-demand dynamic streaming over HTTP, point the `source` property of the VideoDisplay or VideoPlayer class to an *.f4m file:

```
<s:VideoPlayer source="http://mysite.com/dynamic_Streaming.f4m" autoPlay="false"/>
```

HTTP Dynamic Streaming is not designed to be used with the DynamicStreamingVideoSource or the DynamicStreamingVideoItem classes. Use those classes to stream media over RTMP with Flash Media Server.

Before your can dynamically stream your video content over HTTP, use the File Packager to package the file. The File Packager converts the media file to fragments and creates an F4M manifest file to point to the pieces of the package. You must also install the Apache HTTP Origin Module on your web server to serve the streams.

For more information about HTTP Dynamic Streaming, see Adobe HTTP Dynamic Streaming.

Dynamic streaming over RTMP

Flash Media Server 3.5 and later supports live and on-demand dynamic streaming over RTMP. To perform dynamic streaming with Flash Media Server, use the DynamicStreamingVideoSource class.

The DynamicStreamingVideoSource object contains multiple stream items, each of which represents the same video stream encoded at a different bitrate.

The following example uses the DynamicStreamingVideoSource object to define streams of different qualities:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- controls\VideoPlayer\VideoPlayerFMS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:VideoPlayer id="myPlayer"
        width="500" height="300">
        <s:source>
            <s:DynamicStreamingVideoSource id="mySVS"
                host="rtmp://fmsexamples.adobe.com/vod/">
                <s:DynamicStreamingVideoItem id="dreamgirl150"
                    streamName="MP4:_PS_dreamgirl_150.f4v"
                    bitrate="150" />
                <s:DynamicStreamingVideoItem id="dreamgirl500"
                    streamName="MP4:_PS_dreamgirl_500.f4v"
                    bitrate="500" />
                <s:DynamicStreamingVideoItem id="dreamgirl1000"
                    streamName="MP4:_PS_dreamgirl_1000.f4v"
                    bitrate="1000" />
            </s:DynamicStreamingVideoSource>
        </s:source>
    </s:VideoPlayer>
    <s:TextArea width="500" height="50"
        text="Please wait while the video loads..."/>
</s:Application>
```

More Help topics

Dynamic streaming (Flash Media Server documentation)

Adobe HTTP Dynamic Streaming

Playing back live video with the Spark VideoPlayer control

To play back live video, use a DynamicStreamingVideoSource object and set the `streamType` attribute to `live`:

```
<s:VideoPlayer>
    <s:DynamicStreamingVideoSource host="rtmp://localhost/live/" streamType="live">
        <s:DynamicStreamingVideoItem streamName="myStream.flv"/>
    </s:DynamicStreamingVideoSource>
</s:VideoPlayer>
```

The VideoPlayer is the client application that plays back video. To capture and stream the live video, you can use a tool like Flash Media Live Encoder, which captures and streams live video to Flash Media Server. If you don’t use Flash Media Live Encoder, you’ll need to create a custom publishing application. For information on creating a custom video capture application, see the Flash Media Server documentation.

Setting the size of the Spark VideoPlayer control

The appearance of any video media playing in a VideoPlayer control is affected by the following properties:

- `scaleMode`
- `height`
- `width`

The `scaleMode` property only works when you set the `height` or `width` property. It adjusts the size of the playing media after the control size has been set. Possible values are `none`, `stretch`, `letterbox`, and `zoom`.

If you omit both `width` and `height` properties for the control, Flex sizes the control to fit the size of the playing media. If you specify only one property, and set the `scaleMode` property, the size of the playing media determines the value of the other property. If you do not set the `scaleMode` property, the media retains its aspect ratio when resizing.

If you explicitly set the `width` and `height` properties, be sure the values are appropriate for the size of video that plays.

By default, Flex sizes the VideoPlayer control to the size of the media. If you specify the `width` or `height` property of the control, and either is smaller than the media's dimensions, Flex does not change the size of the component. Instead, Flex sizes the media to fit within the component. If the control's playback area is smaller than the default size of the media, Flex shrinks the media to fit inside the control.

The sample video used here is larger than the specified component size, but the width and height ratio are still appropriate to the video size. If you were to remove the width and height properties from the code, however, you would see that the VideoPlayer component is automatically sized to the size of the video.

Using the Spark VideoDisplay control

The Spark VideoDisplay control is the same as the Spark VideoPlayer control, except that it does not have a UI associated with it. You must manually create controls that perform functions such as play, stop, and pause.

The following example uses a Spark VideoDisplay control, and exposes the `play()` and `pause()` methods with Button controls:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\SparkVideoDisplayStopPlay.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="myVid.pause();">
    <s:VGroup>
        <s:VideoDisplay id="myVid"
            source="../assets/MyVideo.flv"
            height="250"
            width="250"
        />
        <s:HGroup>
            <s:Button label="||" click="myVid.pause();"/>
            <s:Button label="&gt;" click="myVid.play();"/>
        </s:HGroup>
    </s:VGroup>
</s:Application>
```

Spark text controls

Text controls in an Adobe® Flex® application can display text, let the user enter text, or do both.

About the Spark text controls

You use Flex text-based controls to display text and to let users enter text into your application.

The following table lists the primary Flex text-based controls:

Control	Component set	Superclass	Multi Line	Editable	textFlow/text/textDisplay
Label	Spark	UIComponent	Y	N	text
RichEditableText	Spark	UIComponent	Y	Y	textFlow/text
RichText	Spark	UIComponent	Y	N	textFlow/text
TextArea	Spark	UIComponent	Y	Y	textFlow/text
TextInput	Spark	UIComponent	N	Y	text/textDisplay (textFlow)

Comparing Spark and MX text controls

This section describes the Spark text controls. For information on the MX text controls, see “MX text controls” on page 805.

How much control you have over the formatting of the text in the control depends on the type of control. The MX controls support a small set of formatting options with their style properties and the `htmlText` property. The Spark controls support a richer set of formatting options because they are based on Flash Text Engine (FTE) and Text Layout Framework (TLF).

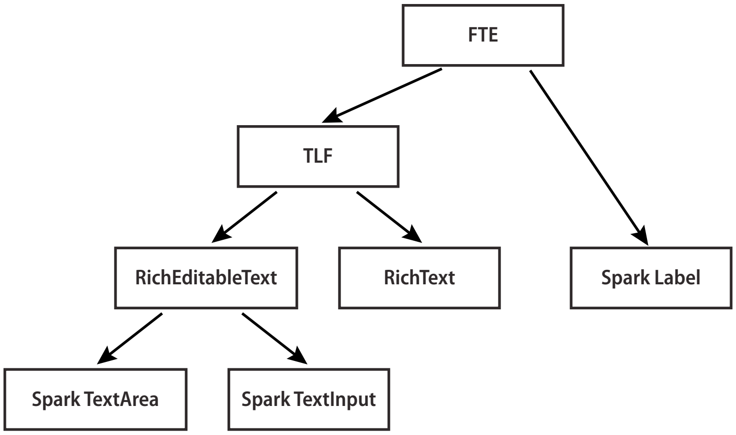
Some MX controls can also use limited functionality of FTE and TLF. For more information, see “Using FTE in MX controls” on page 1594.

When building Flex 4 applications, you should use the Spark text controls where possible. This is especially true if you plan on using TLF or embedded fonts. In some cases, there are both MX and Spark versions of the same text-based control. For example, there are MX and Spark versions of the Label, TextArea, and TextInput controls. The MX versions are in the mx.controls package.*. The Spark versions are in the spark.components.* package.

Spark text control feature overview

The primary Spark controls for displaying text are Label, RichText, and RichEditableText. The Label control has the least amount of functionality, but is also the most lightweight. The RichEditableText control has the most functionality, but also uses the most system resources.

The following image shows the relationship among the Spark text controls and FTE/TLE:



Relationship of the Spark text controls to FTE/TLF

The following table shows more information about these Spark text primitives:

Feature	Spark Label	RichText	RichEditableText
Extends	UIComponent	UIComponent	UIComponent
Uses	FTE	TLF	TLF
Advanced typography	Y	Y	Y
Alpha	Y	Y	Y
Rotation	Y	Y	Y
Bi-directional text	Y	Y	Y
Default formatting with CSS styles	Y	Y	Y
Multiple lines	Y	Y	Y
Multiple formats	N	Y	Y
Multiple paragraphs	N	Y	Y
Text object model	N	Y	Y
Markup language	N	Y	Y
Clickable HTML links	N	N	Y
Inline graphics	N	Y	Y
Hyperlinks	N	N	Y
Scrolling	N	N	Y
Selection	N	N	Y
Editing	N	N	Y

Styling text controls

The default styles for Spark text control are set in CSS. The easiest way to customize your text controls is to use CSS to change style properties. For example, you can adjust the font size and color by using the fontSize and color style properties as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SimpleStyleExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|RichText {
            color: #33CC66;
            fontStyle: italic;
            fontSize: 24;
        }
    </fx:Style>

    <s:RichText id="myRET">
        This is the text.
    </s:RichText>
</s:Application>
```

CSS lets you adjust the styles of all controls of a certain type (using a type selector) or any controls that match a certain class (using a class selector).

Spark text controls have a much richer set of style properties than standard CSS because they support TLF. These style properties include paragraph spacing, line spacing, and margins. The following example sets the line height and space after each paragraph:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/AnotherStyleExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|RichText {
            fontSize: 18;
            lineHeight:16;
            paragraphSpaceAfter: 10;
        }
    </fx:Style>

    <s:RichText id="myRET" width="200">
        <s:p>This is my text. There are many others like it, but this one is mine.</s:p>
        <s:p>My text is my best friend.</s:p>
        <s:p>It is my life. I must master it as I must master my life.</s:p>
        <s:p>Without me, my text is useless.</s:p>
    </s:RichText>
</s:Application>
```

With TLF, you can also manipulate the appearance of text within the TextFlow at compile time or run-time. For more information, see “Styling TLF-based text controls” on page 785.

When styling the TextInput and TextArea controls, you should keep in mind that the text is rendered by a RichEditableText subcomponent. You access properties of this subcomponent with the `textDisplay` property. For more information, see “Subcomponent styles” on page 1527.

Label control

The Label control displays noneditable text. It is the lightest weight of the text controls. The Label control has the following characteristics:

- The user cannot change the text, but the application can modify it.
- You can control the alignment and sizing.
- The control’s background is transparent, so the background of the component’s container shows through.
- The control has no borders, so the label appears as text written directly on its background.
- The control cannot take the focus.
- The control can display multiple lines.
- The control cannot be made selectable.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

To create user-editable text fields, use the TextInput or TextArea controls. For more information, see “TextInput control” on page 740 and “TextArea control” on page 737.

Creating a Label control

You define a Label control in MXML by using the `<s:Label>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkLabelControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="150" height="80">
  <s:Label text="Label1"/>
</s:Application>
```

Sizing a Label control

If you do not specify a width, the Label control automatically resizes when you change the value of the `text` property.

If you explicitly size a Label control so that it is not large enough to accommodate its text, the text is truncated and terminated by an ellipsis (...). The full text displays as a tooltip when you move the mouse over the Label control. If you also set a tooltip by using the `tooltip` property, the tooltip is displayed rather than the text.

TextArea control

The TextArea control is a multiline, editable text field with a border and optional scroll bars. The TextArea control supports the rich text rendering capabilities of Flash Player and AIR. The TextArea control dispatches `change` and `textInput` events.

The TextArea control supports TLF. The text is rendered by a RichEditableText subcomponent that is defined in the TextArea control's skin class.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

To create a single-line, editable text field, use the TextInput control. For more information, see "TextInput control" on page 740. To create noneditable text fields, use the Label control. For more information, see "Label control" on page 737.

To change the disabled color for the TextArea control, you must edit the TextAreaSkin class.

You can set a TextArea control's `editable` property to `false` to prevent editing of the text. You can set a TextArea control's `displayAsPassword` property to conceal input text by displaying characters as asterisks.

Creating a TextArea control

You define a TextArea control in MXML by using the `<s:TextArea>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkTextAreaControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:TextArea id="textConfirm"
        width="300" height="100"
        text="Please enter your thoughts here."/>
</s:Application>
```

Using large amounts of text in the TLF-based TextArea control can cause a degradation in performance. As a result, in some cases, you should use the MX TextArea control. For more information, see "MX TextArea control" on page 827.

Sizing the TextArea control

The TextArea control does not resize to fit the text that it contains.

If the new text exceeds the capacity of a TextArea control and the `horizontalScrollPolicy property` is set to `true` (the default value), the control adds a scrollbar.

Using TLF with the TextArea control

The TextArea control supports the advanced text formatting and manipulation of TLF. You add TLF content to the TextArea control by using either the `textFlow` or `content` properties.

The following example adds a block of TLF-formatted text to the TextArea control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- sparktextcontrols/TextAreaTLF.mxml -->
<s:Application name="RichEditableTextExample"
        xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Panel title="TextArea TLF"
            width="90%" height="90%"
            horizontalCenter="0" verticalCenter="0">
            <s:TextArea id="ta1" textAlign="left" percentWidth="90">
                <s:textFlow>
                    <s:TextFlow>
                        <s:p fontSize="24">TextArea with TLF block</s:p>
                       <s:p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</s:p>
                       <s:p>2) Cras posuere posuere sem, <s:span fontWeight="bold">eu congue
orci mattis quis</s:span>.</s:p>
                        <s:p>3) Curabitur <s:span textDecoration="underline">pulvinar
tellus</s:span> venenatis ipsum tempus lobortis.<s:br/>
                          <s:span color="0x6600CC">Vestibulum eros velit</s:span>, bibendum
at aliquet ut.
                        </s:p>
                    </s:TextFlow>
                </s:textFlow>
            </s:TextArea>
    </s:Panel>
</s:Application>
```

For more information about using TLF with Spark text controls, see “Using Text Layout Framework” on page 758.

Styling the TextArea control

The TextArea control uses a RichEditableText control as a subcomponent to render the text. You can set the values of inheritable style properties on the TextArea contorl, and those properties are inherited by the RichEditableText subcomponent.

To access the RichEditableText control, you use the `textDisplay` property. You can access this property to set style properties that are noninheritable; for example, the `columnCount` and `columnGap` properties.

There are some exceptions to noninheritable styles. For example, the `verticalAlign`, `lineBreak`, and `paddingBottom/Left/Right/Top` style properties are noninheritable. You can set these properties, which are defined on the RichEditableText control, directly on the TextArea control because the TextAreaSkin class passes them through to the subcomponent.

The following example sets styles on the RichEditableText subcomponent with the `textDisplay` property, and also sets the values of some noninheriting style properties that are not normally accessible. You must cast the `textDisplay` property as a RichEditableText object before you can call methods such as `setStyle()` on it.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkTextAreaStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            /*
            Non-inheriting styles you must set on textDisplay:
                columnCount
                columnGap
                columnWidth

            Non-inheriting styles that you can set on TextArea because
            they are passed to the subcomponent through the TextAreaSkin class:
                lineBreak
                paddingTop/Bottom/Left/Right
                verticalAlign
            */

            import spark.components.RichEditableText;

            private function initApp():void {
                RichEditableText(ta1.textDisplay).setStyle("columnCount", 3);
                RichEditableText(ta1.textDisplay).setStyle("columnWidth", 100);
                RichEditableText(ta1.textDisplay).setStyle("columnGap", 15);
            }
        ]]>
    </fx:Script>
    <s:TextArea id="ta1" height="100" width="400" verticalAlign="bottom" paddingBottom="20">
       This is a text area control. Because the text rendering is done by a RichEditableText
subcontrol,
      you have to use the textDisplay property to set the values of some non-inheriting styles.
       Other non-inheriting styles are defined in the skin class and are passed through to the
        subcomponent.
        For inheriting styles, they are inherited by the RichEditableText subcontrol.
    </s:TextArea>
</s:Application>
```

TextInput control

The Spark TextInput control is a single-line text field that is optionally editable. This control supports TLF. The text is rendered by a RichEditableText subcomponent in its skin class.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

To create a multiline, editable text field, use a TextArea control. For more information, see “TextArea control” on page 737. To create noneditable text fields, use the Label or RichText controls. For more information, see “Label control” on page 737 and “RichText control” on page 743.

The TextInput control does not include a label, but you can add one by using a Label control or by nesting the TextInput control in a FormItem container in a Form layout container.

To change the disabled color for the Spark TextInput control, you must edit the TextInputSkin class. The following example shows the differences between the enabled and disabled versions of the Spark and MX TextInput controls:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TextInputDisabledColors.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:VGroup>
        <mx:TextInput enabled="false" text="disabled MX"/>
        <mx:TextInput enabled="true" text="enabled MX"/>
        <s:TextInput enabled="false" text="disabled Spark"/>
        <s:TextInput enabled="true" text="enabled Spark"/>
    </s:VGroup>
</s:Application>
```

You can set a TextInput control's `editable` property to `false` to prevent users from editing the text. You can set a TextInput control's `displayAsPassword` property to conceal the input text by displaying characters as asterisks.

Creating a TextInput control

You define a TextInput control in MXML by using the `<s:TextInput>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkTextInputControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <s:TextInput id="text1" width="100"/>
</s:Application>
```

You can use the `text` property to specify a string of raw text in the TextInput control.

Sizing a TextInput control

If you do not specify a width, the TextInput control is automatically sized when the application first completes loading. The size of the TextInput control does not change in response to user input or programmatic text input.

The TextInput control determines the value of its `measuredWidth` property by using a `widthInChars` property rather than measuring the text assigned to it, because the text frequently starts out empty. This property is calculated based on the current font size and style.

The value of its `measuredHeight` property is determined by the height of the current font. If you change the font style or size, then the TextInput control resizes to accomodate the new settings, even if there is no text in the control at the time of the resizing.

Binding to a TextInput control

In some cases, you might want to bind a variable to the `text` property of a TextInput control so that the control represents a variable value, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkBoundTextInputControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script><![CDATA[
     [Bindable]
     public var myProp:String="This is the initial String myProp.";
  ]]></fx:Script>
  <s:TextInput text="{myProp}"/>
</s:Application>
```

In this example, the TextInput control displays the value of the `myProp` variable. Remember that you must use the `[Bindable]` metadata tag if the variable changes value and the control must track the changed values; also, the compiler generates warnings if you do not use this metadata tag.

Using TLF with the TextInput control

You can use TLF with a TextInput control by accessing the `textDisplay` property and casting it as a RichEditableText object. This property points to the underlying RichEditableText object that renders the text in the TextInput control. The RichEditableText object is a subcomponent that is defined in the TextInput control's skin class.

The following example defines a TextFlow as XML, and assigns that to the TextInput control's `textDisplay`:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TextInputTLF.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Declarations>
        <fx:XML id="myXML">
            <div>
                <p>Hello <span fontWeight='bold'>world</span>!</p>
            </div>
        </fx:XML>
    </fx:Declarations>
    <fx:Script>
        import spark.utils.TextFlowUtil;
        import spark.components.RichEditableText;
        private function initApp():void {
            RichEditableText(text1.textDisplay).textFlow = TextFlowUtil.importFromXML(myXML);
        }

    </fx:Script>
    <s:TextInput id="text1" width="100"/>
</s:Application>
```

For more information about using TLF with Spark text controls, see "Using Text Layout Framework" on page 758.

RichText control

The RichText control is a middleweight Spark text control. It can display richly-formatted text, with multiple character and paragraph formats. However, it is non-interactive: it does not support hyperlinks, scrolling, selection, or editing. If you want a control that supports formatted text plus scrolling, selection, and editing, you can use the RichEditableText control.

For specifying the text, the RichText control supports the `textFlow`, `text`, and `content` properties. If you set the `text` property, the contents are read in as a String; tags such as `<p>` and `<img>` are ignored. If you set the `textFlow` or `content` properties, then the contents are parsed by TLF and stored as a TextFlow object. Tags such as `<p>` and `<img>` are mapped to instances of the ParagraphElement and InlineGraphicElement classes.

To create a RichText control, you use the `<s:RichText>` tag in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/RichTextExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- You can display simple text with the text property. -->
    <s:RichText text="Hello World!"/>
    <!-- Or the text child tag. -->
    <s:RichText>
        <s:text>
            Hello World!
        </s:text>
    </s:RichText>
    <!-- You can display formatted text with the default property. -->
    <s:RichText>
        Hello <s:span fontSize='16'>BIG NEW</s:span> World!
    </s:RichText>
    <!-- Or the TextFlow child tag. -->
    <s:RichText>
        <s:textFlow>
            <s:TextFlow>
                Hello <s:span fontSize='16'>BIG NEW</s:span> World!
            </s:TextFlow>
        </s:textFlow>
    </s:RichText>
</s:Application>
```

The text in a RichText control can be horizontally and vertically aligned but it cannot be scrolled. The contents of the control wraps at the right edge of the control's bounds. If the content extends below the bottom, it is clipped. It is also clipped if you turn off wrapping by setting the `lineBreak` style property to `explicit` and add content that extends past the right edge of the control.

The contents of a RichText control can be exported to XML using the `export()` method, which produces XML. For an example of using the `export()` method, see http://blog.flexexamples.com/2009/07/25/exporting-a-textflow-object-in-flex-4/.

For more information about using TLF with Spark text controls, see "Using Text Layout Framework" on page 758.

RichEditableText control

The RichEditableText is similar to the RichText control in that it can display richly-formatted text, with multiple character and paragraph formats. In addition, the RichEditableText control is interactive: it supports hyperlinks, scrolling, selection, and editing.

The RichEditableText class is similar to the TextArea control, except that it does not have a border, scroll bars, or focus glow.

Creating a RichEditableText control

To create a RichEditableText control in MXML, you use the `<s:RichEditableText>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/RichEditableTextExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:RichEditableText height="100" width="200">
        <s:text>
            Hello World!
        </s:text>
    </s:RichEditableText>
</s:Application>
```

For specifying the text, the RichEditableText control supports the `textFlow`, `text`, and `content` properties. If you set the `text` property, the contents are read in as a String; tags such as `<p>` and `<img>` are ignored. If you set the `textFlow` or `content` properties, then the contents are parsed by TLF and stored in a TextFlow object. Tags such as `<p>` and `<img>` are mapped to instances of the ParagraphElement and InlineGraphicElement classes.

Inserting and appending text

The RichEditableText control also supports the `insertText()` and `appendText()` methods. These methods let you add text to the contents of the control as if you had typed it. The `insertText()` method either replaces a selection (if there is one) or adds the new text at the current insertion point. The `appendText()` adds the text to the end of the contents. New text added with these methods is not parsed by the TLF parser; it is read in as a literal string.

The `insertText()` method does not insert text unless the RichEditableText control has focus. The `appendText()` appends text even if the RichEditableText control does not have the focus. If the RichEditableText control gets focus by pressing the Tab key, the `insertText()` method inserts the text before any existing text. If the control gets focus by clicking within the control, the text is inserted at the insertion point.

The following example lets you append or insert new text to the RichEditableText control:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/AddTextExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
        [Bindable]
        private var newText:String = "<p>This is the new text.</p>";

        private function insertNewText():void {
            myRET.insertText(newText);
        }

        private function appendNewText():void {
            myRET.appendText(newText);
        }
      ]]>
  </fx:Script>

    <s:RichEditableText id="myRET" height="100" width="200">
        <s:textFlow>
            <s:TextFlow>
                <s:p>This is paragraph 1.</s:p>
                <s:p>This is paragraph 2.</s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:RichEditableText>
    <s:HGroup>
        <s:Button label="Insert New Text" click="insertNewText()"/>
        <s:Button label="Append New Text" click="appendNewText()"/>
    </s:HGroup>
    <s:Label text="New text to add: '{newText}'"/>
</s:Application>
```

The contents of a RichEditableText control can be exported to XML using the `export()` method, which produces XML. For an example of using the `export()` method, see http://blog.flexexamples.com/2009/07/25/exporting-a-textflow-object-in-flex-4/.

Making content selectable and editable

You can make the content of the RichEditableText control selectable by setting the `selectable` property to `true`. You can make the content of the RichEditableText control editable by setting the `editable` property to `true`.

The following example makes the RichEditableText content both selectable and editable:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/RETEditableAndSelectable.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:RichEditableText height="100" width="200"
        editable="true"
        selectable="true">
        <s:text>
            This text is editable and selectable!
        </s:text>
    </s:RichEditableText>
</s:Application>
```

You can read the selection range with the read-only `selectionAnchorPosition` and `selectionActivePosition` properties. You can set the selection with the `selectRange()` method. For examples of using the `selectRange()` method, see "Selecting text" on page 754.

Using the getFormatOfRange() and setFormatOfRange() methods

To determine the text formatting on the selection, use the `getFormatOfRange()` method. This method returns a TextLayoutFormat object. This returns the computed value of every property on the control, including defaults.

You can set the format of the selection with the `setFormatOfRange()` method. In general, you should not use `getFormatOfRange()` method to get the TextLayoutFormat and then set properties on that. This overwrites custom properties you might have already set on the text control with the default values. The following example is not a best practice, as it overwrites any custom properties:

```
var tf:TextLayoutFormat = myTextControl.getFormatOfRange(null,0,0);
tf.fontSize = 16;
tf.fontWeight = FontWeight.BOLD;
myTextControl.setFormatOfRange(tf,0,0);
```

The following example uses the `setFormatOfRange()` method to apply properties without overwriting other custom properties with the defaults:

```
var tf:TextLayoutFormat = new TextLayoutFormat();
tf.fontSize = 16;
tf.fontWeight = FontWeight.BOLD;
myTextControl.setFormatOfRange(tf,0,0);
```

Clipping content

You can clip contents of the RichEditableText control and let users scroll it by setting the `clipAndEnableScrolling` property to `true`, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/RETClipAndScroll.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:RichEditableText height="100" width="200" clipAndEnableScrolling="true">
        <s:text>
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
            Hello World!
        </s:text>
    </s:RichEditableText>
</s:Application>
```

Adding scrollbars

The RichEditableText control does not add scroll bars for you, even when scrolling and clipping is enabled. To add scroll bars to a RichEditableText control, you can wrap the RichEditableText control in a Scroller class, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/ScrollableRET.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:Scroller>
        <s:RichEditableText height="100" width="200"
            editable="true"
            selectable="true">
            <s:text>
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
                This text scrolls vertically!
            </s:text>
        </s:RichEditableText>
    </s:Scroller>
    <s:Label text="To enable horizontal scrolling, set lineBreak to explicit:"/>
    <s:Scroller>
        <s:RichEditableText height="100" width="200"
            editable="true"
            selectable="true"
            lineBreak="explicit">
            <s:text>
                This text scrolls horizontally! This text scrolls horizontally! This text 
scrolls horizontally!
                This text scrolls horizontally! This text scrolls horizontally! This text 
scrolls horizontally!
            </s:text>
        </s:RichEditableText>
    </s:Scroller>
</s:Application>
```

The RichEditableText control supports programmatic scrolling with its IViewport interface; it scrolls in response to the mousewheel; and it automatically scrolls as you drag-select or type more text than fits in the control's bounds. The Scroller class lets you put anything that implements the IViewport interface, such as the RichEditableText control, in it to be scrolled.

Selecting ranges

If you select a range of text that is not in the current viewport, you can programmatically scroll to that range by using the `scrollToRange()` method. The following example selects the 10th line and then scrolls to that line when you click the button:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SelectScrollableRET.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function scrollToLine():void {
                myRET.selectRange(440, 471);
                myRET.scrollToRange(440, 471);
            }
        ]]>
    </fx:Script>

    <s:Scroller>
        <s:RichEditableText id="myRET"
            selectionHighlighting="always"
            selectable="true"
            height="100" width="200">
            <s:text>
                This text scrolls vertically1!
                This text scrolls vertically2!
                This text scrolls vertically3!
                This text scrolls vertically4!
                This text scrolls vertically5!
                This text scrolls vertically6!
                This text scrolls vertically7!
                This text scrolls vertically8!
                This text scrolls vertically9!
                This text scrolls vertically10!
                This text scrolls vertically11!
                This text scrolls vertically12!
                This text scrolls vertically13!
                This text scrolls vertically14!
            </s:text>
        </s:RichEditableText>
    </s:Scroller>
    <s:Button label="Select and Scroll to Line 10" click="scrollToLine()"/>
</s:Application>
```

For more information about using TLF with Spark text controls, see “Using Text Layout Framework” on page 758.

Adding content to text controls

The Spark text-based controls let you set and get text by using the following properties:

text Plain text without formatting information. All text based controls support this property. For information on using the `text` property, see “Using the text property” on page 751.

default Setting the default property is the same as using the `textFlow` property, and is less verbose. Tags set in the default property are parsed at compile time, just as any other MXML tag.

textFlow A rich set of tags and TLF formatting. Most Spark text-based controls support this property. You typically use this property if you do not know the contents of the rich text at compile time. For information on using the `textFlow` property, see "Creating TextFlow objects" on page 760.

Typically, if your text does not contain any formatting, use the `text` property. If your text contains formatting, use the `textFlow` or default properties.

You can set formatted text by using the `textFlow` or default properties, and get it back as a plain text string by using the `text` property.

If you set more than one content-related property on a Spark text control in ActionScript (such as `text`*and*`textFlow`), the last one set wins. If you set more than one of these properties as an attribute on an MXML tag, however, the winning setter cannot be predicted. The compiler does not guarantee the order in which MXML attributes get applied to a component.

Using the default property

You can use the default property (`content`) to add text to TLF-based text controls. These controls include the RichText, RichEditableText, and TextArea controls. Text that is added with the default property is parsed by the TLF and stored as a TextFlow object.

The following example sets the default property:

```
<s:RichText id="myRT" width="450">
    This is text.
</s:RichText>
```

If you are using TLF with your text controls, then you should generally use the `textFlow` property to set the contents. The default property is set only, and it is less efficient than the `textFlow` property. If you set the contents of a text control with the default property, you can then get it with either the `textFlow` property (gets a TextFlow object) or the `text` property (gets a plain text String object). For more information about using the `textFlow` property, see "Creating TextFlow objects" on page 760.

Typically, you start the value of the `content` property with a paragraph tag. That tag then contains `<span>` elements or breaks or anchor tags. TLF renders text set by the default property by putting it into a span, and that into a paragraph, and that into a TextFlow, and then rendering the TextFlow. The resulting tree uses the following structure:

```
<TextFlow>
    <p>
        <span>
            <contents ... >
```

When using the default property to add text to a control, you can use a subset of HTML in the content. This list matches the supported tags of the `<s:textFlow>` tag. For a list of supported tags, see "Markup tags supported in TextFlow objects" on page 766.

The content of the default property is typed as Object because you can set it to a String, a FlowElement, or an Array of Strings and FlowElement objects.

In the following example, the `<span>` tag is one FlowElement, and the character data "`World`" is treated as if it were wrapped in a `<String>` tag:

```
<s:RichText fontSize="12" xmlns="library://ns.adobe.com/flex/spark">
    <span fontWeight="bold">Hello</span>World
</s:RichText>
```

You can also set the content to a single FlowElement:

```
<s:RichText fontSize="12" xmlns="library://ns.adobe.com/flex/spark">
    <span fontWeight="bold">Hello World</span>
</s:RichText>
```

or to a single String:

```
<s:RichText fontSize="12" fontWeight="bold">Hello World</s:RichText>
```

Note that if you use a tag for the default property (such as `<span>`), you must specify the namespace in the opening tag of the text control. Otherwise, the compiler will not recognize the tag in the text.

Because there is only one format across all of the content, however, it is more efficient to set the formatting on the RichText tag and the content with the `text` property, as the following example shows:

```
<s:RichText fontSize="12" fontWeight="bold" text="Hello World"/>
```

There is no getter for the default property. Instead, you access the host component's `textFlow` property to get the parsed contents or the component's `text` property to get a simple string.

You can add special characters such as ampersands and angle brackets in the default property. To do this, use the same rules as described in "Specifying special characters in the text property" on page 752.

Using the text property

You can use the text property to specify the text string that appears in a text control or to get the text in the control as a plain text String. All text-based Flex controls support a `text` property. When you set this property, any tags (such as HTML) in the text string appear in the control as literal text.

The interpretation of \r, \n and \r\n as paragraph separators is the only interpretation that occurs when setting the value of the `text` property.

You cannot specify text formatting when you set the `text` property, but you can format the text in the control, as the following example shows:

```
<s:RichText fontSize="12" fontWeight="bold" text="Hello World"/>
```

For Spark controls that support TLF, you can set the text and then manipulate it with the TLF API. You can access nodes of the text object model and format parts of the content.

The following code line uses a `text` property to specify Label text:

```
<s:Label text="This is a simple text label"/>
```

You can also use a `<s:text>` child tag to specify text, as the following example shows:

```
<s:Label>
    <s:text>
    This is a simple text label.
    </s:text>
</s:Label>
```

The way you specify special characters, including quotation marks, greater than and less than signs, and apostrophes, depends on whether you use them in MXML tags or in ActionScript. It also depends on whether you specify the text directly or wrap the text in a CDATA section.

Note: *If you specify the value of the* `text` *property by using a string directly in MXML, Flex collapses white space characters. If you specify the value of the* `text` *property in ActionScript, Flex does not collapse white space characters.*

Specifying special characters in the text property

The following rules specify how to include special characters in the `text` property of a text control MXML tag, either in a property assignment, such as `text="the text"`, or in the body of a `<s:text>` subtag.

In standard text The following rules determine how you use special characters if you do not use a CDATA section:

- To use the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), insert the XML character entity equivalents of `&lt;`, `&gt;`, and `&amp;`, respectively. You can also use `&quot;` and `&apos;` for double-quotation marks (") and single-quotation marks ('), and you can use numeric character references, such as &#165 for the Yen mark (¥). Do not use any other named character entities; Flex treats them as literal text.
- You cannot use the character that encloses the property text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence `\"` for any double-quotation marks in the string. If you surround the string in single-quotation marks (') use the escape sequence `\'` for any single-quotation marks in the string. You *can* use single-quotation marks inside a string that is surrounded in double-quotation marks, and double-quotation marks inside a string that is surrounded in single-quotation marks.
- Flex text controls ignore escape characters such as \t or \n in the `text` property. They ignore or convert to spaces, tabs and line breaks, depending on whether you are specifying a property assignment or a `<s:text>` subtag. To include line breaks, put the text in a CDATA section.

The following code example uses the `text` property with standard text:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkStandardText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="400">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Label width="400"
    text="This string contains a less than, &lt;,
    greater than, &gt;, ampersand, &amp;, apostrophe, ', and
    quotation mark &quot;."/>
  <s:Label width="400"
    text='This string contains a less than, &lt;,
    greater than, &gt;, ampersand, &amp;, apostrophe, &apos;, and
    quotation mark, ".'/>
  <s:Label width="400">
     <s:text>
        This string contains a less than, &lt;, greater than,
        &gt;, ampersand, &amp;, apostrophe, ', and quotation mark, ".
     </s:text>
  </s:Label>
</s:Application>
```

The resulting application contains three almost identical text controls, each with the following text. The first two controls, however, convert any tabs in the text to spaces. The third one has leading and trailing newlines and converts any tabs to newlines.

```
This string contains a less than, <, greater than, >, ampersand, &,apostrophe, ', and quotation
mark, ".
```

In a CDATA section If you wrap the text string in the CDATA tag, the following rules apply:

- You cannot use a CDATA section in a property assignment statement in the text control opening tag; you must define the property in a <s:text> child tag.
- Text inside the CDATA section appears as it is entered, including white space characters. Use literal characters, such as " or < for special characters, and use standard return and tab characters. Character entities, such as &gt;, and backslash-style escape characters, such as \n, appear as literal text.

The following code example follows these CDATA section rules:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkTextCDATA.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="500">

  <s:Label width="100%">
     <s:text>
        <![CDATA[
            This string contains a less than, <, greater than, >,
            ampersand, &, apostrophe, ', return,
            tab. and quotation mark, ".
        ]]>
     </s:text>
  </s:Label>
</s:Application>
```

The displayed text appears on three lines, as follows:

```
    This string contains a less than, <, greater than, >,
    ampersand, &, apostrophe, ', return,
    tab. and quotation mark, ".
```

Specifying special characters in ActionScript

The following rules specify how to include special characters in a text control when you specify the control's text property value in ActionScript; for example, in an initialization function, or when assigning a string value to a variable that you use to populate the property:

- You cannot use the character that encloses the text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence \" for any double-quotation marks in the string. If you surround the string in single-quotation marks ('), use the escape sequence \' for any single-quotation marks in the string.
- Use backslash escape characters for special characters, including \t for the tab character, and \n or \r for a return/line feed character combination. You can use the escape character \" for the double-quotation mark and \' for the single-quotation mark.
- In standard text, but not in CDATA sections, you can use the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), by inserting the XML character entity equivalents of &lt;, &gt;, and &amp;, respectively. You can also use &quot; and &apos; for double-quotation marks ("), and single-quotation marks ('), and you can use numeric character references, such as &#165; for the Yen mark (¥). Do not use any other named character entities; Flex treats them as literal text.
- In CDATA sections only, do not use character entities or references, such as &lt; or &#165; because Flex treats them as literal text. Instead, use the actual character, such as <.

The following example uses an initialization function to set the `text` property to a string that contains these characters:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkInitText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initText()">

  <fx:Script>
     public function initText():void {
        //The following is on one line.
        myText.text="This string contains a return, \n, tab, \t, and quotation mark, \". " +
            "This string also contains less than, &lt;, greater than, &gt;, " +
            "ampersand, &amp;, and apostrophe, ', characters.";
     }
  </fx:Script>
  <s:RichText width="450" id="myText" tabStops="100 200 300 400"/>
</s:Application>
```

The following example uses an `<fx:Script>` tag with a variable in a CDATA section to set the `text` property:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkVarText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            [Bindable]
            //The following is on one line.
          public var myText:String ="This string contains a return, \n, tab, \t, and quotation
mark, \". This string also contains less than, <, greater than, <, ampersand, <;, and
apostrophe, ', characters.";
        ]]>
    </fx:Script>
    <s:RichText width="450" text="{myText}" tabStops="100 200 300 400"/>

</s:Application>
```

The displayed text for each example appears on three lines. The first line ends at the return specified by the \n character. The remaining text wraps onto a third line because it is too long to fit on a single line. (Note: When you specify a tab character, you should be sure to create tab stops with the tabStops style property. Otherwise, the tab will instead be rendered as a carriage return. The RichText, RichEditableText, and Spark TextArea controls support the tabStops property, but the Label control does not.)

```
This string contains a return,
, tab,                   , and quotation mark, ". This string also contains less than, <,
greater than, >, ampersand, &, and apostrophe, ', characters.
```

Selecting text

The editable text controls provide properties and methods to select text regions and get the contents of the selections.

Creating a selection

The following controls let you select text:

- RichEditableText
- TextInput
- TextArea
- All controls that use these controls as a subcomponent

The selectable text controls provide the `selectRange()` method, which selects a range of text. You specify the zero-based indexes of the start character and the position immediately *after* the last character you want to select.

The following example shows how to select the first 10 characters of various text controls:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SetSelectionTest.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function selectText():void {
                sparkTextArea.selectRange(0, 10);
                sparkTextInput.selectRange(0, 10);
                sparkRET.selectRange(0, 10);
            }
        ]]>
    </fx:Script>
    <s:TextArea id="sparkTextArea"
        selectionHighlighting="always"
        selectable="true"
        text="Spark TextArea control."/>
    <s:TextInput id="sparkTextInput"
        selectionHighlighting="always"
        selectable="true"
        text="Spark TextInput control."/>
    <s:RichEditableText id="sparkRET"
        selectionHighlighting="always"
        selectable="true"
        text="Spark RichEditableText control."/>

    <s:Button click="selectText()" label="Select Text"/>

</s:Application>
```

To determine when the selection is highlighted, you can use the `selectionHighlighting` property. Possible values are `always`, `whenFocused`, and `whenActive`.

To select all text in a text control, use the `selectAll()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SelectAllExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function selectText(e:Event):void {
                e.currentTarget.selectAll();
            }
        ]]>
    </fx:Script>
    <s:TextArea id="sparkTextArea"
        text="Spark TextArea control." focusIn="selectText(event)"/>
    <s:TextInput id="sparkTextInput" text="Spark TextInput control."
focusIn="selectText(event)"/>

</s:Application>
```

TLF-based text controls have additional APIs that let you select text in different ways. For example, you can programmatically select a particular SpanElement object within a TextFlow, depending on where the cursor is. To do this, you walk the text object model's tree of elements (or leaves) and compare the cursor's position against each leaf element's character range.

The following example selects only the SpanElement that is under the mouse click:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkSelectionExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.elements.TextRange;
            import flashx.textLayout.elements.*;
            private function selectSomeText(e:Event):void {
                /* Get the location of the cursor. This is the character position of the
                   cursor in the RichEditableText control after the user clicks on it. */
                var activePos:int = richTxt1.selectionActivePosition;
                /* Get the first SpanElement in the TextFlow. */
                var leaf:SpanElement = new SpanElement();
                leaf = SpanElement(richTxt1.textFlow.getFirstLeaf());

                /* Get the start and end index values for the first SpanElement. */
                var spanStart:int = leaf.getParagraph().parentRelativeStart;
                var spanEnd:int = leaf.getParagraph().parentRelativeEnd;

                /* For the first SpanElement, if the cursor position falls within the
                   SpanElement's character range, then select the entire SpanElement. */
                if (activePos >= spanStart && activePos <= spanEnd) {
                    selectSpan(spanStart, spanEnd);
                    return;
                }
                /* Perform the same operations for each leaf in the TextFlow. */
                while(leaf = SpanElement(leaf.getNextLeaf())) {
                    spanStart = leaf.getParagraph().parentRelativeStart;
                    spanEnd = leaf.getParagraph().parentRelativeEnd;
                    if (activePos >= spanStart && activePos <= spanEnd) {
                        selectSpan(spanStart, spanEnd);
                        return;
                    }
                }
            }
            private function selectSpan(i1:int, i2:int):void {
                richTxt1.selectRange(i1, i2);
            }

        ]]>
    </fx:Script>

    <s:Panel>
        <s:RichEditableText id="richTxt1" click="selectSomeText(event)" selectable="true"
editable="true" textAlign="justify" percentWidth="100">
            <s:textFlow>
                <s:TextFlow>
                    <s:p><s:span>1) Lorem ipsum dolor sit amet, consectetur adipiscing
elit.</s:span></s:p>
                    <s:p><s:span>2) Cras posuere posuere sem, eu congue orci mattis
quis.</s:span></s:p>
                    <s:p><s:span>3) Curabitur pulvinar tellus venenatis ipsum tempus
lobortis.</s:span></s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
    </s:Panel>
</s:Application>
```

Rather than iterating over all the leaf elements in the TextFlow object, you can also use the TextFlow class's `findLeaf()` method. This method lets you find an element based on a relative position. For an example of using the `findLeaf()` method, see "Navigating TextFlow objects" on page 767.

Getting a selection

You get the anchor and active positions of the selected text to get a text control's current selection. You then get the value of the characters between those positions.

To get the selected text in a text control, you use the `selectionAnchorPosition` and `selectionActivePosition` properties of the text control. These properties define the position at the start of the selection and at the end of the selection, respectively. You use the `substring()` method on the contents of the control, and pass these positions to identify the range of characters to return.

The following example displays the selections of the RichEditableText control when you click the button:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SparkTraceSelectionRanges.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.core.IUITextField;
            import flashx.textLayout.elements.TextRange;
            private function initApp():void {
                sparkRET.selectRange(0, 10);
            }

            private function getTextSelection():void {
                var anchorPos:int = sparkRET.selectionAnchorPosition;
                var activePos:int = sparkRET.selectionActivePosition;
                myLabel.text = "Current Selection: \"" + sparkRET.text.substring(anchorPos, 
activePos) + "\"";
            }
        ]]>
    </fx:Script>
    <s:RichEditableText id="sparkRET"
        selectionHighlighting="always"
        selectable="true"
        text="Spark RichEditableText control."/>

    <s:Button click="getTextSelection()" label="Show Current Selection"/>

    <s:Label id="myLabel"/>

</s:Application>
```

Using Text Layout Framework

The Text Layout Framework (TLF) is a class library built on top of the Flash Text Engine (FTE). The FTE, available in Flash Player 10 and Adobe AIR 1.5, adds advanced text capabilities to Flash Player. FTE provides a set of low-level APIs for libraries that leverage these capabilities. The FTE classes are in the flash.text.engine package. In most cases, you will not use these classes directly.

The TLF classes are in the flashx.textLayout.* package. You are likely to use the TLF classes when you apply styles with HTML markup to text in text controls that support TLF. These controls include the RichText and RichEditableText controls, as well as the Spark TextArea control.

The primary difference between most Spark and MX text-based controls is the control over the formatting. For Spark controls, the formatting is provided by either FTE or TLF. This gives you a great deal of control over the text formatting, including support for HTML tags, columns, and bi-directional text.

TLF is not used by Flex text controls in mobile applications. When you create a mobile application, the mobile theme is automatically applied. As a result, text-based controls use the mobile skins, which do not support TLF. You can still use some of the same text controls, but be aware that TLF is not used when the mobile theme is applied. For more information on using TLF-based text controls with mobile applications, see Use text in a mobile application.

More Help topics

"Flow" TLF markup application

Features of TLF

Text Layout Framework (TLF) is a class library built on top of FTE. It provides high level text functionality, including:

- Bi-directional text, vertical text, and over 30 writing systems, including Arabic, Hebrew, Chinese, Japanese, Korean, Thai, Lao, and the major writing systems of India
- Selecting, editing, and flowing text across multiple columns and linked containers, as well as around inline images
- Vertical text, Tate-Chu-Yoko (horizontal within vertical text), and justifiers for East Asian typography
- Rich typographical controls, including kerning, ligatures, typographic case, digit case, digit width, and discretionary hyphens
- Cut, copy, paste, undo, and standard keyboard and mouse controls for editing
- Rich developer APIs to manipulate text content, layout, and markup and to create custom text components
- Robust list support including custom markers and numbering formats
- Inline images and positioning rules

When you add text to a control that uses TLF, the text is stored as a hierarchical tree of objects. The root element of this tree is the TextFlow class. Each node in the tree is an instance of a class defined in the flashx.textLayout.elements package. This tree is known as the text object model. The text object model is a subset of the FXG specification.

In the text object model, concepts such as paragraphs, spans, and hyperlinks are not represented as formats that affect the appearance of character runs in a single, central text string. Instead they are represented by runtime-accessible ActionScript objects, with their own properties, methods, and events (in some cases).

The following lists the objects you can have in a TextFlow object:

- Paragraphs (ParagraphElement)
- Images (InlineGraphicElement)
- Hyperlinks (LinkElement)
- Spans (SpanElement)
- TCY blocks (TCYElement)
- Tabs (TabElement) and line breaks (BreakElement)
- Lists (ListElement)

If a Spark text control supports the `textFlow` property, then you can use TLF to format and programmatically interact with the contents of that text control. If the control supports only the `text` property for its content, then you can only use a simple string for the text. The text does not get parsed into the text object model. If an MX control supports the `htmlText` property, then you can use a subset of HTML tags to format the text, but its content is still a simple string that does not support the formal text object model.

You use the `textFlow` property when you set the value of the text at run time. The content of the `textFlow` property is strongly typed as a TextFlow object rather than an Object.

You use the default property when you set the value of the property at compile time. This property takes a generic Object and converts it to a TextFlow. In general, you should avoid using the default property when the `textFlow` property is available.

The Spark versions of the TextArea and TextInput classes support TLF, as do the RichText and RichEditableText controls. The Spark Label control supports FTE. The default MX controls do not support TLF. However, it is possible to use TLF with some MX controls. For more informations, see “Using FTE in MX controls” on page 1594.

For detailed information about using TLF and advanced typography, see Basics of Working with text.

Creating TextFlow objects

The text object model is the tree that is created when you add content to a text-based control that supports TLF. This tree is defined by the TextFlow class.

When creating a TextFlow object, you can use the classes in the flashx.textLayout.elements.* package to provide formatting and additional functionality. These classes include DivElement, ParagraphElement, InlineGraphicElement, LinkElement, SpanElement, TabElement, BreakElement, and TCYElement.

You can create a TextFlow object in the following ways:

- Using the default property
- Using the `<s:textFlow>` child tag
- Importing content with the TextFlowUtil class
- Using ActionScript

Using the default property

You can create a TextFlow object by adding content to a text-based’s default property. When you do this, Flex creates a TextFlow object that defines the text object model for you. For more information, see “Using the default property” on page 750.

Creating a TextFlow object with child tags

You can also use the text control’s `<s:textFlow>` child tag to explicitly create a TextFlow object. This lets you structure the contents of the text control by using the supported TLF tags such as `<p>`, `<div>` and `<span>`.

When using the `<s:textFlow>` child tag, you must be sure to also include a `<s:TextFlow>` tag within that, so that the structure of the control’s uses the following structure:

```
<s:text_control>
    <s:textFlow>
        <s:TextFlow>
            <.../>
        </s:TextFlow>
    </s:textFlow>
</s:text_control>
```

You should keep in mind that the default property can also create a TextFlow object, so you could write the previous example as follows:

```
<s:text_control>
    <.../>
</s:text_control>
```

The following example creates two text controls; one text control defines a TextFlow object with the `<s:textFlow>` child tag, and the other with the default property.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/CreateTextFlowChildTags.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:RichEditableText id="richTxt1" textAlign="justify" percentWidth="100">
        <s:textFlow>
            <s:TextFlow>
                <s:p fontSize="24">TextFlow Child Tag</s:p>
                <s:p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</s:p>
                <s:p>2) Cras posuere posuere sem, eu congue orci mattis quis.</s:p>
                <s:p>3) Curabitur pulvinar tellus venenatis ipsum tempus lobortis.</s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:RichEditableText>
    <s:RichEditableText id="richTxt2" textAlign="justify" percentWidth="100">
        <s:p fontSize="24">Default Property</s:p>
        <s:p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</s:p>
        <s:p>2) Cras posuere posuere sem, eu congue orci mattis quis.</s:p>
        <s:p>3) Curabitur pulvinar tellus venenatis ipsum tempus lobortis.</s:p>
    </s:RichEditableText>
</s:Application>
```

It is also convenient to drop the prefix on the markup tags by making the Spark library namespace the default namespace on the text control's tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TextFlowNamespace.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:RichEditableText id="richTxt1" textAlign="justify" percentWidth="100"
xmlns="library://ns.adobe.com/flex/spark">
        <textFlow>
            <TextFlow>
                <p fontSize="24">TextFlow Child Tag</p>
                <p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
                <p>2) Cras posuere posuere sem, eu congue orci mattis quis.</p>
                <p>3) Curabitur pulvinar tellus venenatis ipsum tempus lobortis.</p>
            </TextFlow>
        </textFlow>
    </s:RichEditableText>
</s:Application>
```

Creating a TextFlow object with the TextFlowUtil class

If you don’t know the rich text that you want to display until run-time, you can import HTML or XML to a TextFlow object. The TextFlowUtil class has the following methods that let you add XML and strings as the contents of a TextFlow object:

- `importFromString()`
- `importFromXML()`

The following example uses the `importFromString()` method to load a String object into the TextFlow:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TextFlowMarkup.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="doSomething()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import flashx.textLayout.elements.TextFlow;
        import spark.utils.TextFlowUtil;

        private function doSomething():void {
            var markup:String = "<p>This is paragraph 1.</p><p>This is paragraph 2.</p>";
            var flow:TextFlow = TextFlowUtil.importFromString(markup);
            myST.textFlow = flow;
        }
    ]]></fx:Script>
    <s:RichText id="myST" width="175"/>
</s:Application>
```

You can add also import XML by using the TextFlowUtil class’s `importFromXML()` method. Any XML that uses valid TLF markup can be passed into a TextFlow. This includes content returned by an HTTPService call, XML defined in the application file itself, or XML defined in an external file.

The following example uses an external file that defines a TextFlow object for a text control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- sparktextcontrols/ExternalXMLFile.mxml -->
<s:Application name="Spark_RichText_text_test"
        xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import spark.utils.TextFlowUtil;

            XML.ignoreWhitespace = false;
        ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:XML id="textFlowAsXML" source="externalTextFlow1.xml" />
    </fx:Declarations>

    <s:RichText id="richTxt"
            textFlow="{TextFlowUtil.importFromXML(textFlowAsXML)}"
            horizontalCenter="0" verticalCenter="0" />

</s:Application>
```

The previous example uses the following file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- http://blog.flexexamples.com/2009/08/11/setting-text-in-a-spark-richtext-control-in-
flex-4/ -->
<TextFlow xmlns="http://ns.adobe.com/textLayout/2008" whiteSpaceCollapse="preserve">
    <p><span>The quick brown </span> <span fontWeight="bold">fox jumps over</span> <span> the
lazy dogg.</span></p>
</TextFlow>
```

If the first tag in an imported XML object is not a `<s:TextFlow>` tag, then the parser inserts one for you. If the first tag is a `<s:TextFlow>` tag, then you must also define the `"http://ns.adobe.com/textLayout/2008"` namespace in that tag.

The parser converts the TLF tags to TLF classes (such as SpanElement or DivElement) in the new TextFlow object.

This example used with permission from http://blog.flexexamples.com.

Creating a TextFlow object with the TextConverter class

You can use the TextConverter class to import content into a TextFlow. This is a little more advanced than using the TextFlowUtil methods to create a TextFlow object. Those methods act as wrappers around the TextConverter class. When you import text with the TextConverter class's `importToFlow()` method, however, you can specify the format of the text. This determines how the content is stored.

The following are the types of formats you can specify when using the `importToFlow()` method:

- HTML
- Plain text
- TLF

If you specify the TLF format, then the imported text is used to generate a text object model. If you specify plain text, then the text is not interpreted, but instead stored as a simple text string. If you specify HTML format, then the HTML tags in the text are mapped to TLF classes (for example, `<p>` is mapped to a ParagraphElement object in the text object model).

The following example imports an XML object into a TextFlow when you click the Button control:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/AddingContentAsXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import spark.utils.TextFlowUtil;
        import flashx.textLayout.elements.*;
        import flashx.textLayout.conversion.*;
        import mx.utils.*;
        private var xml:XML = <TextFlow
xmlns='http://ns.adobe.com/textLayout/2008'><p><span>This is a span</span></p></TextFlow>;

        private function addContent():void {
           myRET.textFlow= TextConverter.importToFlow(xml, TextConverter.TEXT_LAYOUT_FORMAT);
        }
        ]]>
    </fx:Script>

    <s:RichEditableText id="myRET" height="100" width="200"/>

    <s:Button click="addContent()" label="Add Content"/>
</s:Application>
```

The following example uses a variety of formats when importing a String into the TextFlow with the `importToFlow()` method:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- sparksparktextcontrols/ImportToFlowExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import flashx.textLayout.conversion.TextConverter;

            XML.ignoreWhitespace = false;
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <!-- Define a String to use with HTML and plain text format. -->
        <fx:String id="htmlTextAsHTML"><![CDATA[<p>Hello <b>world!</b></p>]]></fx:String>

        <!-- Define an XML object to use with TLF format. -->
        <fx:XML id="tfTextAsTextFlow">
            <TextFlow xmlns="http://ns.adobe.com/textLayout/2008">
                <p>Hello <span fontWeight="bold">world!</span></p>
            </TextFlow>
        </fx:XML>
    </fx:Declarations>

    <s:RichText id="richTxt"
            textFlow="{TextConverter.importToFlow(htmlTextAsHTML,
TextConverter.TEXT_FIELD_HTML_FORMAT)}"
            horizontalCenter="0" verticalCenter="0" />

     <s:RichText id="richTxt2"
             textFlow="{TextConverter.importToFlow(htmlTextAsHTML,
TextConverter.PLAIN_TEXT_FORMAT)}"
             horizontalCenter="0" verticalCenter="0" />

     <s:RichText id="richTxt3"
             textFlow="{TextConverter.importToFlow(tfTextAsTextFlow,
TextConverter.TEXT_LAYOUT_FORMAT)}"
             horizontalCenter="0" verticalCenter="0" />

</s:Application>
```

When you use the `TEXT_LAYOUT_FORMAT` type for the TextConverter, the imported text must observe the following rules:

1 The first tag must be a `<s:TextFlow>` tag. The parser will not insert one for you as it does with the `TextFlowUtil.importFromXML()` or `TextFlowUtil.importFromString()` methods.

2 The first tag must specify a namespace for the imported text. The namespace is `"http://ns.adobe.com/textLayout/2008"`.

3 Subsequent tags must be supported by the TextFlow class. For a list of supported tags, see "Markup tags supported in TextFlow objects" on page 766.

The TextConverter class also lets you export a TextFlow object's content in a variety of formats. For an example of a text exporter, see http://blog.flexexamples.com/2009/07/25/exporting-a-textflow-object-in-flex-4/.

Creating a TextFlow object in ActionScript

You can create TextFlow objects that are used by TLF-based text controls in ActionScript.

If you create a TextFlow object in ActionScript, the TextFlow object requires that either the ParagraphElement or DivElement be at the top level. The following example creates two ParagraphElement objects and wraps them in SpanElement objects. It then adds these objects as children of a TextFlow object, and adds them to a RichEditableText control's content:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SimpleTextModelExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import flashx.textLayout.elements.*;
        private var textFlow:TextFlow = new TextFlow();
        private var paragraph1:ParagraphElement = new ParagraphElement();
        private var paragraph2:ParagraphElement = new ParagraphElement();
        private var span1:SpanElement = new SpanElement();
        private var span2:SpanElement = new SpanElement();

        private function initApp():void {
            span1.text = "This is paragraph one in myRET.";
            span2.text = "This is paragraph two in myRET.";
            paragraph1.addChild(span1);
            paragraph2.addChild(span2);
            textFlow.addChild(paragraph1);
            textFlow.addChild(paragraph2);

            myRET.textFlow = textFlow;
        }
    </fx:Script>

    <s:RichEditableText id="myRET" height="100" width="200"/>
</s:Application>
```

Markup tags supported in TextFlow objects

The TLF parser converts supported markup elements to TLF classes so that they can be used in the text object model. These markup elements are similar to the HTML tags of the same names. For example, if you specify a `<p>` tag in your text control's content, then the parser converts it to a ParagraphElement in the control's TextFlow.

The following table describes the supported elements of the text object model:

Element	Class	Description
div	DivElement	A division of text; can contain only div, list, or p elements.

Element	Class	Description
p	ParagraphElement	A paragraph; can contain any element except div, list, or li.
a	LinkElement	A hypertext link, also known as an anchor; can contain the tcy, span, img, tab, g, and br elements. This is the only class in the text object model that supports Flex events.
tcy	TCYElement	A run of horizontal text, used in vertical text such as Japanese; can contain the a, span, img, tab, g, or br elements.
span	SpanElement	A run of text in a paragraph; cannot contain other elements.
img	InlineGraphicElement	An image in a paragraph element.
tab	TabElement	A tab character.
br	BreakElement	A break character; text continues on the next line, but does not start a new paragraph.
list	ListElement	An ordered or unordered list. Can be simple list or nested. You can customize the bullets. Can contain li elements (ListItemElement) or other list elements.
li	ListItemElement	List items within a list element. Can be ordered or unordered, nested or flat lists. You can customize the markers as well as other settings of each list item.
g	SubParagraphGroupElement	A group element. Used for grouping elements in a paragraph. This lets you nest elements below the paragraph level.

Navigating TextFlow objects

The TLF API provides methods and properties that let you navigate the text object model after it is created. You can use this API to perform actions on a individual leaves or the entire tree.

The following example iterates over the RichEditableText control’s TextFlow and copies the contents of each SpanElement, one at a time, into another RichEditableText control:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/IterateOverLeaves.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.elements.*;
            private function copyContents():void {
                /* Get the first leaf in the TextFlow. */
                var leaf:SpanElement = new SpanElement();
                leaf = SpanElement(richTxt1.textFlow.getFirstLeaf());

                /* Write the contents of the first leaf to the second RET control. */
                richTxt2.text = "LEAF:" + leaf.text + "\n";
                /* Iterate over the remaining leaves and write their contents
                   to the second RET control. */
                while(leaf = SpanElement(leaf.getNextLeaf())) {
                    richTxt2.text += "LEAF:" + leaf.text + "\n";
                }
            }
        ]]>
    </fx:Script>

    <s:Panel>
      <s:RichEditableText id="richTxt1" selectable="true" editable="true" textAlign="justify" 
percentWidth="100">
            <s:textFlow>
                <s:TextFlow>
                    <s:p><s:span>1) Lorem ipsum dolor sit amet, consectetur adipiscing 
elit.</s:span></s:p>
                    <s:p><s:span>2) Cras posuere posuere sem, eu congue orci mattis 
quis.</s:span></s:p>
                    <s:p><s:span>3) Curabitur pulvinar tellus venenatis ipsum tempus 
lobortis.</s:span></s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
    </s:Panel>
    <s:HGroup>
        <s:Button label="Copy Contents" click="copyContents()"/>
        <s:Button label="Clear" click="richTxt2.text=''"/>
    </s:HGroup>
    <s:Panel>
        <s:RichEditableText id="richTxt2" textAlign="justify" percentWidth="100"/>
    </s:Panel>
</s:Application>
```

The `TextFlow.findLeaf()` method lets you specify a character position and returns the entire flow element where that character position occurs. The following example changes the color of the entire SpanElement whenever the user clicks on any location within that SpanElement's paragraph:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/FindLeafExample.mxml -->
<s:Application 
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.elements.TextRange;
            import flashx.textLayout.elements.*;
            private function selectEntireLeaf(e:Event):void {
                /* Get the location of the cursor. This is the character position of the 
                   cursor in the RichEditableText control after the user clicks on it. */
                var activePos:int = richTxt1.selectionActivePosition;
                /* Change the color of the entire leaf under the cursor position. */
                var leaf:SpanElement = richTxt1.textFlow.findLeaf(activePos) as SpanElement;
                leaf.color = 0x00FF33;
            }
        ]]>
    </fx:Script>

    <s:Panel>
        <s:RichEditableText id="richTxt1" click="selectEntireLeaf(event)" selectable="true" 
editable="true" textAlign="justify" percentWidth="100">
            <s:textFlow>
                <s:TextFlow>
                    <s:p><s:span>1) Lorem ipsum dolor sit amet, consectetur adipiscing 
elit.</s:span></s:p>
                    <s:p><s:span>2) Cras posuere posuere sem, eu congue orci mattis 
quis.</s:span></s:p>
                    <s:p><s:span>3) Curabitur pulvinar tellus venenatis ipsum tempus 
lobortis.</s:span></s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
    </s:Panel>
</s:Application>
```

You can access the objects in a TextFlow by their object names and set the values of properties such as `color` or `direction` on them in ActionScript. The text in the controls update accordingly, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/ManipulateTextModelExample.mxml -->
<s:Application 
    xmlns:fx="http://ns.adobe.com/mxml/2009" 
    xmlns:mx="library://ns.adobe.com/flex/mx" 
    xmlns:s="library://ns.adobe.com/flex/spark" 
    creationComplete="initApp()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import flashx.textLayout.elements.*;
        private var textFlow:TextFlow = new TextFlow();
        private var paragraph1:ParagraphElement = new ParagraphElement();
        private var paragraph2:ParagraphElement = new ParagraphElement();
        private var span1:SpanElement = new SpanElement();
        private var span2:SpanElement = new SpanElement();

        private function initApp():void {
            span1.text = "This is paragraph one.";
            span2.text = "This is paragraph two.";
            paragraph1.addChild(span1);
            paragraph2.addChild(span2);
            textFlow.addChild(paragraph1);
            textFlow.addChild(paragraph2);

            myRET.textFlow = textFlow;
        }

        private function changeColors():void {
            // Change color of first paragraph.
            paragraph1.color = 0xFF00FF;

            // Change color of second paragraph.
            paragraph2.setStyle("color", 0x00FF00);
        }
    </fx:Script>

    <s:RichEditableText id="myRET" height="100" width="200">
    </s:RichEditableText>

    <s:Button label="Change Colors" click="changeColors()"/>
</s:Application>
```

This example shows that you can set styles by using either the `setStyle()` method or the property accessors. For more information, see “Styling TLF-based text controls” on page 785.

You can also access the objects in the text object model by using their ids. You do this with the TextFlow class’s `getElementById()` method. This requires that you set the `id` property of the flow elements, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/AccessTextFlowMethods.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import flashx.textLayout.elements.*;
        private var textFlow:TextFlow = new TextFlow();
        private var paragraph1:ParagraphElement = new ParagraphElement();
        private var paragraph2:ParagraphElement = new ParagraphElement();
        private var span1:SpanElement = new SpanElement();
        private var span2:SpanElement = new SpanElement();

        private function initApp():void {
            span1.id = "span1";
            span2.id = "span2";
            paragraph1.id = "paragraph1";
            paragraph2.id = "paragraph2";

            span1.text = "This is paragraph one.";
            span2.text = "This is paragraph two.";
            paragraph1.addChild(span1);
            paragraph2.addChild(span2);
            textFlow.addChild(paragraph1);
            textFlow.addChild(paragraph2);

            myRET.textFlow = textFlow;
        }

        private function changeColors():void {
            // Set color of paragraph one.
            textFlow.getElementByID("paragraph1").setStyle("color", 0xFF0000);
            // Set color of paragraph two.
        textFlow.getElementByID("paragraph2").setStyle("color", 0xFF0000);
        }
    </fx:Script>

    <s:RichEditableText id="myRET" height="100" width="200">
    </s:RichEditableText>

    <s:Button label="Change Colors" click="changeColors()"/>
</s:Application>
```

In the previous example, because the text flow elements are created programmatically, their `id` properties are also set explicitly on the objects that represent the span and paragraph elements. If you were creating the text flow with HTML markup, you would specify the value of the `id` property in the HTML tag (for example, `<span id="span1">`).

Another way to “walk the tree” of a TextFlow object is to use the methods of the ParagraphElement class. This class helps you access low-level items such as atoms and words. For example, you can use the `findNextWordBoundary()` and `findPreviousWordBoundary()` methods to navigate the word boundaries in the flow element’s text.

The following example uses methods of the ParagraphElement class to count the words and determine the average length of them in the TextFlow:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/FindWords.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.elements.*;
            private function countWords():void {
                var leaf:SpanElement = new SpanElement();
                leaf = SpanElement(richTxt1.textFlow.getFirstLeaf());

                var p:ParagraphElement = new ParagraphElement();
                p = leaf.getParagraph();

                doSomething(p);
                while (leaf = SpanElement(leaf.getNextLeaf())) {
                    p = leaf.getParagraph();
                    doSomething(p);
                }
                wcLabel.text += "# Words: " + wordCount;
           lenLabel.text += "Avg length of each word: " + lenTotal/wordCount + " chars";
            }
            private var wordCount:int = 0;
            private var lenTotal:int = 0;
            private function doSomething(p:ParagraphElement):void {
                var wordBoundary:int = 0;
                var prevBoundary:int = 0;

                // If these are equal, then there are no more words.
                while (wordBoundary != p.findNextWordBoundary(wordBoundary)) {
                    if (p.findNextWordBoundary(wordBoundary) - wordBoundary > 1) {
                        wordCount += 1;
                    }
                    prevBoundary = wordBoundary;
                    wordBoundary = p.findNextWordBoundary(wordBoundary);
                  // If the value is greater than 1, then it's a word, otherwise it's a space.
                    if (wordBoundary - prevBoundary > 1) {
                        var s:String = p.getText().substring(prevBoundary, wordBoundary);
                        lenTotal += s.length;
                    }
                }
            }
        ]]>
    </fx:Script>

    <s:Panel>
      <s:RichEditableText id="richTxt1" selectable="true" editable="true" textAlign="justify"
percentWidth="100">
            <s:textFlow>
                <s:TextFlow>
                    <s:p><s:span>Lorem ipsum dolor sit amet, consectetur adipiscing
elit.</s:span></s:p>
                    <s:p><s:span>Cras posuere posuere sem, eu congue orci mattis
quis.</s:span></s:p>
                    <s:p><s:span>Curabitur pulvinar tellus venenatis ipsum tempus
lobortis.</s:span></s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
    </s:Panel>
    <s:VGroup>
        <s:Button label="Count" click="countWords()"/>
        <s:Label id="wcLabel"/>
        <s:Label id="lenLabel"/>
    </s:VGroup>
</s:Application>
```

Adding images with TLF

TLF supports embedding images in text controls by using the InlineGraphicElement class. To add an image to the text object model, use the `<img>` tag in a `<s:textFlow>` tag or the default property, or create an instance of the InlineGraphicElement class.

The InlineGraphicElement class can point to an image file, such as a GIF, JPG, or PNG file.

You specify the location of the image by using the `source` property. The location can be relative to the deployed location of the application (for example, “images/butterfly.gif”) or it can be a full path to the image (for example, “http://yourserver.com/images/butterfly.gif”).

The following example loads a simple image from a local location:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SimpleInlineGraphic.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="doSomething()">
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.elements.*;
            import flashx.textLayout.*;

            [Bindable]
            private var textFlow:TextFlow;
            private var img:InlineGraphicElement;

            private function doSomething():void {
                textFlow = new TextFlow();
                var p:ParagraphElement = new ParagraphElement();
                img = new InlineGraphicElement();
                img.source = "assets/butterfly.gif";
                img.height = 100;
                img.width = 100;
                p.addChild(img);
                textFlow.addChild(p);
            }
            ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Simple Inline Graphic Image"
        width="90%" height="90%"
        horizontalCenter="0" verticalCenter="0">
        <s:RichEditableText id="richTxt" textAlign="justify" width="100%"
            textFlow="{textFlow}" />
    </s:Panel>
</s:Application>
```

As with text-based examples, you can use a variety of techniques to load the image with an InlineGraphicElement object. The following example uses child tags:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SimpleInlineGraphicTags.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Simple Inline Graphic Image"
        width="90%" height="90%"
        horizontalCenter="0" verticalCenter="0">
        <s:RichEditableText id="richTxt" textAlign="justify" width="100%">
            <s:textFlow>
                <s:TextFlow>
                    <s:p>
                       <s:img source="@Embed(source='../assets/butterfly.gif')" height="100"
width="100"/>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
    </s:Panel>
</s:Application>
```

Rather than load the image at run time, you can also use an embedded image for the InlineGraphicImage object's source, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SimpleEmbed.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        [Embed(source="../assets/butterfly.gif")]
        [Bindable]
        public var imgCls:Class;
    </fx:Script>
        <s:RichEditableText id="richTxt" textAlign="justify" width="100%">
            <s:textFlow>
                <s:TextFlow>
                    <s:p>
                        <s:img id="myImage" source="{imgCls}" height="100" width="100"/>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>

</s:Application>
```

The InlineGraphicElement class can also display a drawn element, such as a Sprite or an FXG component. The following example loads an FXG component into the InlineGraphicElement:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/FXGInlineGraphic.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*"
    creationComplete="doSomething()">
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.elements.*;
            import flashx.textLayout.*;
            import comps.*;

            [Bindable]
            private var textFlow:TextFlow;
            private var img:InlineGraphicElement;

            private function doSomething():void {
                textFlow = new TextFlow();
                var p:ParagraphElement = new ParagraphElement();
                img = new InlineGraphicElement();
                img.source = new ArrowAbsolute();
                img.height = 100;
                img.width = 100;
                p.addChild(img);
                textFlow.addChild(p);
            }
            ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="FXG Inline Graphic Image"
        width="90%" height="90%"
        horizontalCenter="0" verticalCenter="0">
        <s:RichEditableText id="richTxt" textAlign="justify" width="100%"
            textFlow="{textFlow}" />
    </s:Panel>
</s:Application>
```

The FXG component used in this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- textcontrols/comps/ArrowAbsolute.fxg -->
<fxg:Graphic xmlns:fxg="http://ns.adobe.com/fxg/2008" version="1">
    <!-- Use Use compact syntax with absolute coordinates. -->
    <fxg:Path data="
         M 20 0
         C 50 0 50 35 20 35
         L 15 35
         L 15 45
         L 0 32
         L 15 19
         L 15 29
         L 20 29
         C 44 29 44 6 20 6
    ">
         <!-- Define the border color of the arrow. -->
         <fxg:stroke>
              <fxg:SolidColorStroke color="#888888"/>
         </fxg:stroke>
         <!-- Define the fill for the arrow. -->
         <fxg:fill>
              <fxg:LinearGradient rotation="90">
                   <fxg:GradientEntry color="#000000" alpha="0.8"/>
                   <fxg:GradientEntry color="#FFFFFF" alpha="0.8"/>
              </fxg:LinearGradient>
         </fxg:fill>
    </fxg:Path>
</fxg:Graphic>
```

For more information about using FXG, see “FXG and MXML graphics” on page 1714.

If you do not know the image’s dimensions, you can use the InlineGraphicElement class’s `measuredHeight` and `measuredWidth` properties to define the size of the image. You can only use these properties after the image loads. To do this, you can trigger a function that sizes the image off of the TextFlow object’s `StatusChangeEvent` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/LoadImageEvent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="initApp()">
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.events.StatusChangeEvent;
            import flashx.textLayout.elements.*;
            import flashx.textLayout.*;

            [Bindable]
            private var textFlow:TextFlow;
            private var img:InlineGraphicElement;

            private function initApp():void {
                textFlow = new TextFlow();
                textFlow.addEventListener(StatusChangeEvent.INLINE_GRAPHIC_STATUS_CHANGE,
sizeGraphic);
                var p:ParagraphElement = new ParagraphElement();
                img = new InlineGraphicElement();
                img.source = "assets/butterfly.gif";
                p.addChild(img);
                textFlow.addChild(p);
            }
            private function sizeGraphic(e:StatusChangeEvent):void {
                if (e.status == "ready" || e.status == "sizePending") {
                    img.height = img.measuredHeight;
                    img.width = img.measuredWidth;
                }
            }
            ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Sizing Inline Graphic Image"
        width="90%" height="90%"
        horizontalCenter="0" verticalCenter="0">
        <s:RichEditableText id="richTxt" textAlign="justify" width="100%"
            textFlow="{textFlow}" />
    </s:Panel>
</s:Application>
```

You can get a reference to the InlineGraphicElement's graphic by using the `graphic` property. This property points to the embedded DisplayObject. This lets you interact with the InlineGraphicElement in the same ways that you can interact with any DisplayObject. For example, you can apply blend modes, change the alpha, rotate, and scale the image.

The following example changes the `alpha` of the DisplayObject when you move the slider:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/DisplayObjectImageElement.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        private function changeAlpha():void {
            myImage.graphic.alpha = .5;
        }
    </fx:Script>
        <s:RichEditableText id="richTxt" textAlign="justify" width="100%">
            <s:textFlow>
                <s:TextFlow>
                    <s:p>
                        <s:img id="myImage"
                            source="@Embed(source='../assets/butterfly.gif')"
                            height="100" width="100"/>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>

        <s:HSlider id="hSlider"
            minimum="0" maximum="1"
            value="1"
            stepSize="0.1"
            snapInterval="0.1"
            liveDragging="true"
            valueCommit="myImage.graphic.alpha=hSlider.value;"/>

</s:Application>
```

Note that the previous example uses a RichEditableText control and not a RichText control. The RichEditableText control supports user interaction, while the RichText control does not.

Positioning images inline

To position the InlineGraphicElement within a text element, you use the following properties:

- `float` property of the InlineGraphicElement class
- `clearFloats` property of the FlowElement (or the `clearFloats` style property of the text container)
- `paddingTop/paddingBottom/paddingLeft/paddingRight` properties on the FlowElement class

The `float` property controls the placement of the inline graphic within the text. The following example shows a simple image surrounded by text. By changing the value of the `float` property, you can change the way text flows around the image, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFFloat.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function changeSelection(event:Event):void {
                image1.float = event.currentTarget.selectedItem;
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:Label text="float for the image (default:none):"/>
        <s:ComboBox id="cb1"
            change="changeSelection(event)"
            creationComplete="cb1.selectedIndex=0">
            <s:ArrayCollection>
                <fx:String>none</fx:String>
                <fx:String>left</fx:String>
                <fx:String>right</fx:String>
                <fx:String>start</fx:String>
                <fx:String>end</fx:String>
            </s:ArrayCollection>
        </s:ComboBox>
    </s:HGroup>

    <s:RichEditableText id="myRET1" width="300">
        <s:textFlow>
            <s:TextFlow columnWidth="290">
               <s:p id="p1">Images in a flow are a good thing. For example, here is a float.
<s:img id="image1" float="none" source="@Embed(source='../assets/bulldog.jpg')"
paddingRight="10" paddingTop="10" paddingBottom="10" paddingLeft="10"></s:img>Don't you
agree?
                It should show on the left. If it doesn't show up on the left, then it is a
bug. You can submit bugs at http://bugs.adobe.com/jira/. You can set how the float is positioned
within
               the paragraph by using the ComboBox below. You can select left, right, start,
end, or none. This example does not use the clearFloats property. That is in a different
example.</s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:RichEditableText>

</s:Application>
```

You use the constants defined by the flashx.textLayout.formats.Float class to set the value of the `float` property. The values “left” and “right” indicate that the image should be left-aligned or right-aligned within the text. The values of “start” and “end” are similar, in that they left- and right-align the image, but the meanings of them depend on the direction of the text (RTL or LTR). For RTL text, for example, “end” means the end of the line, which is typically the left side and “start” means the start of the line, which is typically the right side. The value of “none” means that no text should flow around the image. The default value is “none”.

The `clearFloats` property controls the placement of the paragraph elements relative to the `float`. For example, if you want to ensure that a second paragraph after a floating image starts below the image, you might set `clearFloats` on the second paragraph to “both” or “left”, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFClearFloats.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:HGroup>
        <s:Label text="clearFloats for second paragraph (default:none):"/>
        <s:ComboBox id="cb1"
            change="p2.clearFloats = cb1.selectedItem"
            creationComplete="cb1.selectedIndex=0">
            <s:ArrayCollection>
                <fx:String>none</fx:String>
                <fx:String>left</fx:String>
                <fx:String>right</fx:String>
                <fx:String>start</fx:String>
                <fx:String>end</fx:String>
                <fx:String>both</fx:String>
            </s:ArrayCollection>
        </s:ComboBox>
    </s:HGroup>
    <s:RichEditableText id="myRET1" width="300" height="400">
        <s:textFlow>
            <s:TextFlow columnWidth="275">
                <s:p id="p0" fontWeight="bold">Heading 1</s:p>
              <s:p id="p1">Images in a flow are a good thing. <s:img id="image1" float="left"
source="@Embed(source='../assets/bulldog.jpg')" paddingRight="10" paddingTop="10"
paddingBottom="10" paddingLeft="10"></s:img> You can use the ComboBox to change the clearFloats
value on the second paragraph ("Heading 2"). This should ensure that Heading 2 starts on a new
line after the image.</s:p>
                <s:p id="p2" fontWeight="bold">Heading 2</s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:RichEditableText>

</s:Application>
```

You use the constants defined by the flashx.textLayout.formats.ClearFloats class to set the value of the `clearFloats` property. The values "left" and "right" indicate that the new paragraph should be placed after any float that appears on the left or right. As with the `float` property, the "start" and "end" values are dependent on the direction of the text in the paragraph. A value of "both" causes text to appear after all floats. A value of "none" lets the text appear next to the float. The default value is "none".

Note that you can also set the `clearFloats` property as a style on the TLF-based text controls such as RichEditableText and Spark TextArea. This style property is inheritable, which means that all child flow elements will inherit the value you set on the control.

To further customize the location of images within text blocks, you can apply the padding properties (`paddingTop`, `paddingBottom`, `paddingRight`, and `paddingLeft`) to any FlowElement object, including an InlineGraphicElement object such as an image. This lets you control the offset of the image from the margin as well as the offset between the image and the text that wraps around it. For more information, see Using padding in TLF.

More Help topics

TLF Floats

Adding hyperlinks with TLF

TLF-based text controls support hyperlinks with the LinkElement class. To insert a hyperlink into the TextFlow, you use the `<a>` tag in a `<s:textFlow>` tag or the default property, or create an instance of the LinkElement class.

You can only use hyperlinks in a TextFlow object in the TextArea and RichEditableText controls. The Spark Label, TextInput, and RichText controls do not support hyperlinks.

The default behavior of a LinkElement object in the text object model is to launch a new browser window and navigate to the location specified in the `href` property of the LinkElement object. The following example shows a basic navigation link:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SimpleLinkElement.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
        <s:RichEditableText id="richTxt"
            editable="false"
            focusEnabled="false">
            <s:textFlow>
                <s:TextFlow>
                    <s:p>
                        The following link takes you to: <s:a
href="http://www.adobe.com">Adobe.com</s:a>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
        <s:RichEditableText id="richTxt2"
            editable="true"
            focusEnabled="false">
            <s:textFlow>
                <s:TextFlow>
                    <s:p>
                        Hold CTRL key down when using the following link: <s:a
href="http://www.adobe.com">Adobe.com</s:a>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>

</s:Application>
```

To ensure that your links use the hand cursor when the user mouses over them, set the `editable` property to `false` on the RichEditableText object. If you do not, then your user must hold the Control key down while moving the cursor over the link to be able to click on that link.

The LinkElement is the only flow element that supports Flex events. Supported events include `click`, `mouseDown`, and `rollOut`.

While the default behavior of the LinkElement object is to launch a new browser window and open the specified link as a new page, you can also define a custom event handler for the `click` event. The easiest way to do this is to exclude the `href` attribute from the anchor tag.

The following example defines custom behavior for the `click` event of the LinkElement object. It does not specify a destination link in the LinkElement, but instead builds one in the `click` event handler:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/CustomLinkElement.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.events.FlowElementMouseEvent;
            import mx.collections.ArrayCollection;

            [Bindable]
            public var myArray:ArrayCollection = new ArrayCollection(["Flex", "Flash",
"ActionScript"]);
            private function handleClickEvent(e:FlowElementMouseEvent):void {
               var url:String = "http://www.google.com/search?q=" + productList.selectedItem;
                navigateToURL(new URLRequest(url), '_blank');
            }
        ]]>
    </fx:Script>
    <s:RichEditableText id="richTxt"
        editable="false"
        focusEnabled="false">
        <s:textFlow>
            <s:TextFlow>
                <s:p>
                 Search for product info on <s:a click="handleClickEvent(event)">Google</s:a>
                </s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:RichEditableText>
    <s:ComboBox id="productList" dataProvider="{myArray}" prompt="Select one"/>

</s:Application>
```

You can also suppress the click event in the event handler by calling the `stopPropagation()` and `preventDefault()` methods in the `click` event's handler.

The `stopPropagation()` method prevents processing of any event listeners in nodes subsequent to the current node in the event flow. The `preventDefault()` method cancels the event's default behavior.

You can then access properties of the LinkElement object to carry out additional actions. The following example presents the user with an option of navigating to the target location or cancelling the action:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/CustomLinkElementHandling.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import flashx.textLayout.events.FlowElementMouseEvent;
        import flashx.textLayout.elements.LinkElement;
        import mx.controls.Alert;
        import mx.events.CloseEvent;

        private var linkTarget:String;

        private function doSomething(e:FlowElementMouseEvent):void {
            e.stopImmediatePropagation();
            e.preventDefault();
            var le:LinkElement = e.flowElement as LinkElement;
            linkTarget = le.href;
            Alert.show("You are about to navigate away from this page.","Alert",Alert.OK |
Alert.CANCEL, this, alertListener, null, Alert.OK);
        }
        private function alertListener(e:CloseEvent):void {
            if (e.detail == Alert.OK) {
                navigateToURL(new URLRequest(linkTarget), '_self')
            }
        }
    </fx:Script>
        <s:RichEditableText id="richTxt"
            editable="false"
            focusEnabled="false">
            <s:textFlow>
                <s:TextFlow>
                    <s:p>
                        The following link takes you to: <s:a href="http://www.adobe.com"
target="_blank" click="doSomething(event)">Adobe.com</s:a>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>

</s:Application>
```

The default styling for a LinkElement is underlined, blue text. To style the links in TLF, you can use a SpanElement object and set style properties on that. The following example removes the underline, and makes the link color red rather than blue:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/StyledLinkElement.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
        <s:RichEditableText id="richTxt"
            editable="false"
            focusEnabled="false">
            <s:textFlow>
                <s:TextFlow>
                    <s:p>
                    The following link takes you to: <s:a href="http://www.adobe.com"><s:span
color="0xFF0066" textDecoration="none">Adobe.com</s:span></s:a>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
</s:Application>
```

You can also use the TextLayoutFormat class to style hyperlinks. For more information, see "Applying styles with the TextLayoutFormat class" on page 789.

More Help topics

TLF FlowElement and LinkElement Events and EventMirrors

Styling TLF-based text controls

The Spark text controls TextInput, TextArea, Label, RichText, and RichEditableText support specifying default text formatting with CSS styles. The complete set of styles supports all of TLF's formatting capabilities, including kerning and bidirectionality.

The names and descriptions of the styles supported by the TLF text controls are the described in the TextLayoutFormat and SelectionFormat classes.

The default values for these styles is defined by the `global` selector in the defaults.css file. This file is compiled into the framework.swc file. To change the values of the defaults, you can create your own global selector; for example:

```
<fx:Styles>
    global {
        fontFamily: "Verdana"
    }
</fx:Style>
```

TLF's FlowElement tags such as `<p>` and `<span>`, which are supported in the content of the RichText or RichEditableText controls, support formatting only with properties, not using CSS. As a result, to set styles on individual flow elements, you must set them as you would set a style property, but not in CSS.

The following example shows setting style properties on the paragraphs. It shows you that you can set the style properties either with the `setStyle()` method or as a property assignment.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFStylesProperties.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import flashx.textLayout.elements.*;

        private function initApp():void {
            // You cannot set these properties in CSS.

            // Set color of first paragraph to red with the setStyle() method.
            myRET.textFlow.getChildAt(0).setStyle("color", 0xFF0000);
            // Set color of second paragraph to green with a property assignment.
            myRET.textFlow.getChildAt(1).color = 0x00FF00;
        }
    </fx:Script>

    <s:RichEditableText id="myRET" height="100" width="200">
        <s:p id="p1">This is paragraph 1.</s:p>
        <s:p id="p2">This is paragraph 2.</s:p>
    </s:RichEditableText>

    <s:Button click="initApp()" label="Apply Styles"/>
</s:Application>
```

The RichText and RichEditableText controls support additional style properties that Label does not. These properties include properties related to columns and paragraphs, including `columnCount`, `columnGap`, `paragraphSpaceAfter`, `paragraphStartIndent`, and `whiteSpaceCollapse`.

The RichEditableText control supports style properties that are not supported by the Label and RichText controls. These properties include `selectionColor`, `inactiveSelectionColor`, and `unfocusedSelectionColor`.

Style inheritance

Some styles are inheriting, meaning that if you set them on a container they affect the children of that container. Generally, the choice of whether a CSS style is inheriting or non-inheriting is made based on whether the corresponding TLF format inherits from the parent FlowElement to the child FlowElement.

The following example shows an inheriting style and a non-inheriting style set on the Button control. The inheriting style, `color`, is applied to the text, but the non-inheriting style, `backgroundColor`, is not.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";

        s|Button {
            color: red;
            backgroundColor: #33CC99;
        }
    </fx:Style>
    <s:Button label="Click Me"/>
</s:Application>
```

Because a skinnable component is a type of Spark container, inheriting styles are inherited by children of the container. If you set an inheriting style such as `fontSize` on a Spark Button, for example, the style is applied to the Label class that renders the Button's label. The Label control is defined in the ButtonSkin class. The parts that make up a larger component are known as subcomponents. If you set a non-inheriting style such as `backgroundColor` it will not affect the Label subcomponent of a Button; instead you can create a custom type selector for Label, or you can reskin the Button and set the value of the `backgroundColor` property directly on the Label.

To set the value of a non-inheriting style property on the Button's label, you can set the value of the `backgroundColor` property in a Label type selector. The following example sets the `backgroundColor` style to #33CC99 for the Label type, which affects the way the Button control's label is rendered because the Label class is a subcomponent of Button:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFStylesSubComps.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";

        s|Button {
            color: red;
        }

        s|Label {
            backgroundColor: #33CC99;
        }
    </fx:Style>
    <s:Button label="Click Me"/>
</s:Application>
```

For the TextArea and TextInput controls, the text is rendered by a RichEditableText subcomponent. You can access the RichEditableText subcomponent by casting the result of the `textDisplay` property to a type that supports the `setStyle()` method. You can then call the `setStyle()` method on this object, which applies non-inheritable style properties to the subcomponent.

Note that this technique does not work for mobile versions of the TextArea and TextInput controls because the mobile skins do not include support for the RichEditableText subcomponent. For more information on using TLF-based text controls with mobile applications, see Text controls.

The following example illustrates the difference between correctly and incorrectly applying a non-inheriting style property to a pair of TextArea controls:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TextAreaStyling.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="doSomething()">
    <fx:Script>
        import spark.components.RichEditableText;

        private function doSomething():void {
            /* To set a non-inheritable style on a TextArea, you must actually
               apply it to the underlying RichEditableText subcomponent, which is
               accessed through the textDisplay property: */
            RichEditableText(text1.textDisplay).setStyle("columnCount", 2);
            /* Setting a non-inheritable style directly on the TextArea does
               not apply the style properly. */
            text2.setStyle("columnCount", 2);
        }

    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="text1" width="200" height="100">
        <s:textFlow>
            <s:TextFlow>
              <s:p>This is TextArea #1. This is enough text to ensure that there will be more
than one column if the columnCount property is properly applied.</s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:TextArea>
    <s:TextArea id="text2" width="200" height="100">
        <s:textFlow>
            <s:TextFlow>
              <s:p>This is TextArea #2. This is enough text to ensure that there will be more
than one column if the columnCount property is properly applied.</s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:TextArea>
</s:Application>
```

You can also change the number of columns in a TextFlow object by setting the `columnCount` property on the `<s:TextFlow>` tag, as the following example shows:

```
<s:TextFlow columnCount="2">
```

Applying styles with the TextLayoutFormat class

To style TLF-based text controls, you can also use the TextLayoutFormat class. You begin by creating an instance of the TextLayoutFormat class. You can then set any formatting properties on that object, including paragraph indentation, leading, justification, and typographic case. Finally, you specify the custom TextLayoutFormat with the TextFlow object's `hostFormat` property.

The following example sets various styles on the text:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TextLayoutFormatExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import flashx.textLayout.formats.*;

        private function initApp():void {
            var textLayoutFormat:TextLayoutFormat = new TextLayoutFormat();
            textLayoutFormat.color = 0x336633;
            textLayoutFormat.fontFamily = "Arial, Helvetica, _sans";
            textLayoutFormat.fontSize = 14;
            textLayoutFormat.paragraphSpaceBefore = 15;
            textLayoutFormat.paragraphSpaceAfter = 15;
            textLayoutFormat.typographicCase = TLFTypographicCase.LOWERCASE_TO_SMALL_CAPS;
            textFlow.hostFormat = textLayoutFormat;

        }
    </fx:Script>
        <s:RichEditableText id="richTxt"
            editable="false"
            focusEnabled="false">
            <s:textFlow>
                <s:TextFlow id="textFlow">
                    <s:p>
                        The following link takes you to <s:a
href="http://www.adobe.com">Adobe.com</s:a>
                    </s:p>
                    <s:p>
                        The following link takes you to <s:a
href="http://www.omniture.com">Omniture.com</s:a>
                    </s:p>
                </s:TextFlow>
            </s:textFlow>
        </s:RichEditableText>
</s:Application>
```

You can define a TextLayoutFormat object inline, rather than in ActionScript. The following example defines TextLayoutFormat objects that style the hyperlink, based on its state:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/StyledLinkElement2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:RichEditableText id="richTxt"
        editable="false"
        focusEnabled="false">
        <s:textFlow>
            <s:TextFlow>
                <s:p>
                    <s:linkHoverFormat>
                        <s:TextLayoutFormat color="#33CC00" textDecoration="underline"/>
                    </s:linkHoverFormat>
                    <s:linkNormalFormat>
                        <s:TextLayoutFormat color="#009900"/>
                    </s:linkNormalFormat>
                    <s:a href="http://www.adobe.com">Adobe.com</s:a>
                </s:p>
            </s:TextFlow>
        </s:textFlow>
    </s:RichEditableText>
</s:Application>
```

Applying styles with the Configuration class

Another approach to styling TLF is to use the Configuration class. This class provides access to some properties of text, such as its state. For example, you can use this class to define the appearance of a hyperlink when it is active or when it is hovered over. You can also use this class to define the appearance of text when it is selected.

To use a Configuration object with TLF, you define the styles on the TextLayoutFormat or SelectionFormat objects and assign them to the Configuration object's format properties.

The format properties include the following:

- `defaultLinkActiveFormat`
- `defaultLinkHoverFormat`
- `defaultLinkNormalFormat`
- `focusedSelectionFormat`
- `inactiveSelectionFormat`
- `textFlowInitialFormat`
- `unfocusedSelectionFormat`

The following example defines the appearance of the hyperlink by using custom TextLayoutFormat objects attached to a Configuration object:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- sparktextcontrols/StylingWithConfig.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               creationComplete="initApp()">
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.conversion.TextConverter;
            import flashx.textLayout.elements.Configuration;
            import flashx.textLayout.elements.IConfiguration;
            import flashx.textLayout.formats.ITextLayoutFormat;
            import flashx.textLayout.formats.TextDecoration;
            import flashx.textLayout.formats.TextLayoutFormat;
            private function initApp():void {
                var txt:String = "Check out our website at <a
href='http://www.adobe.com/'>adobe.com</a>.";
                var cfg:Configuration = new Configuration(true);
                var normalFmt:TextLayoutFormat = new
TextLayoutFormat(cfg.defaultLinkNormalFormat);
                normalFmt.color = 0xFF0000; // red
                normalFmt.textDecoration = TextDecoration.NONE;
            var hoverFmt:TextLayoutFormat = new TextLayoutFormat(cfg.defaultLinkHoverFormat);
                hoverFmt.color = 0xFF00FF; // purple
                hoverFmt.textDecoration = TextDecoration.UNDERLINE;
                cfg.defaultLinkNormalFormat = normalFmt;
                cfg.defaultLinkHoverFormat = hoverFmt;
                rt.textFlow = TextConverter.importToFlow(txt,
TextConverter.TEXT_FIELD_HTML_FORMAT, cfg);
            }
        ]]>
    </fx:Script>

    <s:RichEditableText id="rt" x="20" y="20" editable="false"/>

</s:Application>
```

Skinning TLF-based text controls

You do not typically add skins or chrome to the Spark text controls.

The Label, RichText, and RichEditableText Spark text controls are used in the skins of skinnable components. Because each has a different set of features, you can use the lightest weight text control that meets your needs.

For example, the default skin of a Spark Button uses the Label class to render the label of the Button. If you have a Button that requires rich text, you can sometimes replace the Label control in its skin with a RichText control.

The default skins of the Spark TextInput and TextArea controls use a RichEditableText control to provide an area in which text can be edited. The border and background are provided by a Rect class, and the scrollbars by a Scroller class.

For more information on skinning Spark controls, see “Spark Skinning” on page 1602.

Mirroring and bidirectional text

Some languages, such as Hebrew and Arabic, read from right to left. However, those languages often include text from other languages that read from left to right. Most of a sentence in Hebrew, for example, might read from right to left, but it might include English words that are read from left to right. This makes it necessary for Flex controls to support bidirectional text.

To use bidirectional text in a TLF text-based control, use the `direction` style.

Bidirectional text lets you render text from left-to-right (LTR) or right-to-left (RTL). You can embed RTL text inside an LTR block so that portions of a paragraph render LTR and portions render RTL. There is a dominant direction for the entire paragraph, but parts of the paragraph can be read in the opposite direction, and this can be nested.

The following simple example embeds a small amount of Hebrew text in a text control, and sets the `direction` style to `rtl`. The text is written and stored using the UTF-8 encoding.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- sparktextcontrols/RTLHebrewExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">
        <s:layout><s:VerticalLayout/></s:layout>
    <s:Graphic>
       <s:Label direction="rtl" text="???? ??????? ????? ????? ??????  This is a Hebrew test"
width="100" height="100"/>
    </s:Graphic>
</s:Application>
```

The following example includes English and two examples of RTL text (Hebrew and Arabic). The text is defined as Unicode characters, which makes it easier to follow. You can click the button to toggle the RichEditableText control from RTL to LTR.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/RTLTest.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="addText()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @namespace s "library://ns.adobe.com/flex/spark";
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontFamily;
        advancedAntiAliasing: true;
        embedAsCFF: true;
        unicodeRange:
           U+0041-005A, /* Latin upper-case [A..Z] */
           U+0061-007A, /* Latin lower-case a-z */
           U+002E-002E, /* Latin period [.] */
           U+05E1,      /* The necessary Hebrew letters */
           U+05B5,
           U+05E4,
           U+05B6,
           U+05E8,
           U+0645,      /* The necessary Arabic letters */
           U+062F,
           U+0631,
           U+0633,
           U+0629;
     }
     s|RichText {
        fontFamily: myFontFamily;
        fontSize: 32;
        paddingTop: 10;
     }
  </fx:Style>

  <fx:Script>
  import flashx.textLayout.formats.*;
  import spark.utils.TextFlowUtil;
    private function addText():void {
        myRT.textFlow = TextFlowUtil.importFromString("school is written " +
            String.fromCharCode(0x0645, 0x062f, 0x0631, 0x0633, 0x0629) +
           " in Arabic and " + String.fromCharCode(0x05E1, 0x05B5, 0x05E4, 0x05B6, 0x05E8) +
            " in Hebrew.");
    }
    private function mirrorText():void {
        if (myRT.getStyle("direction")=="ltr") {
            myRT.setStyle("direction", flashx.textLayout.formats.Direction.RTL);
        } else {
            myRT.setStyle("direction", flashx.textLayout.formats.Direction.LTR);
        }
    }
  </fx:Script>
  <s:Panel title="RTL and LTR with embedded font">
        <s:RichText id="myRT" width="400" height="150"/>
  </s:Panel>
  <s:Button click="mirrorText()" label="Toggle Direction"/>
</s:Application>
```

Mirroring refers to laying the chrome of a component, or an entire application, out in one direction and then display in the opposite direction. For example, a mirrored TextArea would have its vertical scrollbar on the left. A mirrored Tree would have its disclosure triangles on the right. A mirrored HGroup would have its first child on the right, and a mirrored TabBar would have its first tab on the right. Mirroring can apply to subcomponents as well as chrome. For example, a mirrored RadioButton would have its label on the left side instead of the right side (the default).

Mirroring of components complement bidirectional text by having the application and components reflect the text direction.

Using numbered and bulleted lists with TLF-based text controls

You can use the ListElement and ListItemElement classes to add bulleted lists to your text controls. The bulleted lists can be nested and can be customized to use different bullets (or markers) and auto-numbering.

To create lists in your applications, use the `<s:list>` tag inside a text control that uses TLF. You then use `<s:li>` tags within the `<s:list>` tag for each list item in the list. You can customize the appearance of the bullets by using the ListMarkerFormat class.

The following example creates simple lists:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFLists.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <!-- Simple bulleted list. -->
    <s:RichEditableText id="myRET1" width="200">
        <s:p>Simple bulleted list:</s:p>
        <s:list id="list1" listStylePosition="outside">
            <s:li>Item 1</s:li>
            <s:li>Item 2</s:li>
            <s:li>Item 3</s:li>
        </s:list>
    </s:RichEditableText>
    <!-- Nested bulleted list. -->
    <s:RichEditableText id="myRET2" width="200">
        <s:p>Nested bulleted list:</s:p>
        <s:list id="list2">
            <s:li>Dry cleaning</s:li>
            <s:li>Groceries
                <s:list id="list1a">
                    <s:li>Quart of milk</s:li>
                    <s:li>Loaf of bread</s:li>
                    <s:li>Stick of butter</s:li>
                </s:list>
            </s:li>
            <s:li>Oil change
                <s:list id="list1b">
                    <s:li>Item 1b</s:li>
                </s:list>
            </s:li>
        </s:list>
    </s:RichEditableText>
    <!-- Custom spacing list. -->
    <s:RichEditableText id="myRET3" width="200">
        <s:p>List with custom indents:</s:p>
        <s:list id="list3" paddingLeft="20">
            <s:li>Dry cleaning</s:li>
            <s:li>Groceries
                <s:list id="list31" paddingLeft="20">
                    <s:li>Quart of milk</s:li>
                    <s:li>Loaf of bread</s:li>
                    <s:li>Stick of butter</s:li>
                </s:list>
            </s:li>
            <s:li>Item 3</s:li>
        </s:list>
    </s:RichEditableText>
    <!-- Custom styled list. -->
    <s:RichEditableText id="myRET4" width="200">
        <s:p>Styled bulleted list:</s:p>
        <s:list id="list4" listStylePosition="inside">
            <s:listMarkerFormat>
                <s:ListMarkerFormat fontSize="14" fontWeight="bold" color="0xFF0066" 
afterContent=" "/>
            </s:listMarkerFormat>
            <s:li>Quart of milk</s:li>
            <s:li>Loaf of bread</s:li>
            <s:li>Stick of butter</s:li>
        </s:list>
    </s:RichEditableText>
</s:Application>
```

To change the type of marker or use numbering in the list, use the `listStyleType` property of the ListElement. This property can be any value defined by the ListStyleType class (such as `check`, `circle`, `decimal`, and `box`). The following example creates lists with various marker types and a custom increment:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFListsCustomBullets.mxml -->
<s:Application 
    xmlns:fx="http://ns.adobe.com/mxml/2009" 
    xmlns:mx="library://ns.adobe.com/flex/mx" 
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Custom bullets lists. -->
    <s:RichEditableText id="myRET1" width="200">
        <s:p>Lists with custom bullets:</s:p>
        <s:p>Box:</s:p>
        <s:list id="list1" listStyleType="box">
            <s:li>Quart of milk</s:li>
            <s:li>Loaf of bread</s:li>
        </s:list>
        <s:p>Decimal:</s:p>
        <s:list id="list2" listStyleType="decimal">
            <s:li>Quart of milk</s:li>
            <s:li>Loaf of bread</s:li>
        </s:list>
        <s:p>Check:</s:p>
        <s:list id="list3" listStyleType="check">
            <s:li>Quart of milk</s:li>
            <s:li>Loaf of bread</s:li>
        </s:list>
        <s:p>Nested Roman:</s:p>
        <s:list id="list4" listStyleType="upperRoman">
            <s:li>Item 1
```

```
                <s:list id="list41" listStyleType="lowerRoman">
                    <s:li>Quart of milk</s:li>
                    <s:li>Loaf of bread</s:li>
                </s:list>
            </s:li>
            <s:li>Item 2</s:li>
        </s:list>
    </s:RichEditableText>

    <!-- List with custom increment. -->
    <s:RichEditableText id="myRET2" width="200">
        <s:p>List with countdown:</s:p>
        <s:list id="list5" listStyleType="decimal">
            <s:listMarkerFormat>
                <!-- Increments the list by 2s rather than 1s. -->
                <s:ListMarkerFormat counterIncrement="ordered 2"/>
            </s:listMarkerFormat>
            <s:li>Quart of milk
                <s:listMarkerFormat>
                    <!-- Causes counter to start at 10. -->
                    <s:ListMarkerFormat counterReset="ordered 9"/>
                </s:listMarkerFormat>
            </s:li>
            <s:li>Loaf of bread</s:li>
            <s:li>Stick of butter</s:li>
        </s:list>
    </s:RichEditableText>
</s:Application>
```

You can further customize the appearance of the markers in your lists by using the ListMarkerFormat class to define "before" and "after" content. This is content that appears before and after the content of the marker.

The following example adds the word "Chapter" before the marker, and a colon after the marker. The marker, or content, is ordered upper-case Roman numerals.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/TLFListsBeforeAfter.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Before/After content. -->
    <s:RichEditableText id="myRET" width="500">
        <s:p>List with Before/After content:</s:p>
        <s:list id="myList" paddingLeft="10" listStylePosition="inside">
            <s:listMarkerFormat>
            <!-- Note that this example inserts an empty string with &quote; HTML entities. -->
                <s:ListMarkerFormat
                    beforeContent="Chapter "
                    content="counters(ordered,&quot;&quot;,upperRoman)"
                    afterContent=": "/>
            </s:listMarkerFormat>
            <s:li>Beginning</s:li>
            <s:li>Middle</s:li>
            <s:li>End</s:li>
        </s:list>
        <s:list id="myList2" paddingLeft="10" listStylePosition="inside">
            <s:listMarkerFormat>
                <s:ListMarkerFormat
                    beforeContent="*BEFORE*"
                    content="counters(ordered,&quot;*SUFFIX*&quot;,upperRoman)"
                    afterContent="*AFTER*"/>
            </s:listMarkerFormat>
            <s:li>Beginning</s:li>
            <s:li>Middle</s:li>
            <s:li>End</s:li>
        </s:list>
    </s:RichEditableText>
</s:Application>
```

As the previous example shows, you can use the `content` property to insert a suffix: a string that appears after the marker, but before the `afterContent`. To insert this string when providing XML content to the flow, wrap the string in `&quote;` HTML entities rather than quotation marks (`"<`*`string`*`>"`). The previous example inserted an empty string.

More Help topics

TLF 2.0 Lists Markup

Using prompts with text controls

The Spark TextInput and TextArea controls support a `prompt` property that lets you specify text that helps users know what to enter. For example, an input field might include prompt text that says "Search".

Prompt text appears when the text control is first created. Prompt text disappears when the control gets focus or when the control's `text` property is non-null. Prompt text reappears when the control loses focus, but only if no text was entered (if the value of the text field is the empty string).

For text controls, if the user enters text, but later deletes it, the prompt text reappears when the control loses focus. You can also cause the prompt text to reappear programmatically by setting the text control's text property to `null` or the empty string.

The following example creates a TextInput and TextArea control with prompt text. You can focus on the text controls and enter text, which causes the prompt text to disappear. If you click the Reset button, the prompt text reappears.

```
<?xml version="1.0"?>
<!-- sparktextcontrols/BasicPromptExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Script>
    <![CDATA[
        /* If you reset the text properties of the TextInput and TextArea controls to null
            or the empty string, the prompt text reappears. */
        private function resetPrompts():void {
            myTextInput.text = "";
            myTextArea.text = "";
        }
      ]]>
  </fx:Script>

    <s:TextInput id="myTextInput" prompt="Enter name..."/>
    <s:TextArea id="myTextArea" prompt="Enter details..."/>
    <s:Button label="Reset" click="resetPrompts()"/>

</s:Application>
```

The Spark ComboBox control also supports the `prompt` property because it uses the Spark TextInput control as a subcontrol. For more information, see "Use prompt text with the Spark ComboBox control" on page 542.

Text prompt styles

The default prompt styles are defined in the current theme's default style sheet. By default, the prompt text is gray and italicised. The style of the prompt text is defined by the text control's pseudo selectors.

Pseudo selectors let you apply styles to a component based on its current state. For example, if your TextInput control has prompt text and is not disabled, the style is defined by the `normalWithPrompt` pseudo selector because it is in the `normalWithPrompt` state. If the control is disabled, then the styles defined by the `disabledWithPrompt` pseudo selector are used.

You can change the prompt text styles by overriding the default CSS. The following example changes the prompt text styles by defining different behavior for the `normalWithPrompt` and `disabledWithPrompt` pseudo selectors for the TextInput and TextArea controls:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/StylingPrompts.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|TextInput:normalWithPrompt {
            color: #CCCCFF;
            fontStyle: italic;
        }
        s|TextArea:normalWithPrompt {
            color: #CCCCFF;
            fontStyle: italic;
        }
        s|TextInput:disabledWithPrompt {
            color: #CCFFFF;
            fontStyle: italic;
        }
        s|TextArea:disabledWithPrompt {
            color: #CCFFFF;
            fontStyle: italic;
        }
    </fx:Style>

  <fx:Script>
    <![CDATA[
        private function resetPrompts():void {
            myTextInput.text = "";
            myTextArea.text = "";
        }

        /* Disabling the controls causes the disabledWithPrompt pseudo selector to be used
            instead of the normalWithPrompt pseudo selector. */
        private function disableControls():void {
            if (myTextInput.enabled) {
                myTextInput.enabled = false;
                myTextArea.enabled = false;
            } else {
                myTextInput.enabled = true;
                myTextArea.enabled = true;
            }
        }
      ]]>
  </fx:Script>

    <s:TextInput id="myTextInput" prompt="Enter name..."/>
    <s:TextArea id="myTextArea" prompt="Enter details..."/>

    <s:HGroup>
        <s:Button label="Reset" click="resetPrompts()"/>
        <s:Button label="Toggle Enabled/Disabled" click="disableControls()"/>
    </s:HGroup>
</s:Application>
```

In general, you should avoid setting the prompt text to be larger or smaller than the regular text in the text control. Unless you explicitly define the height and width of the control, the text control's size will grow or shrink as the size of the text or prompt text grows and shrinks.

To simplify your CSS you can also define text prompt styles on the SkinnableTextBase type selector. Styles set on this selector will apply to the TextArea and TextInput controls, as the following example shows:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/SimplerStylingPrompts.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|SkinnableTextBase:normalWithPrompt {
            color: #CCCCFF;
            fontStyle: italic;
        }
        s|SkinnableTextBase:disabledWithPrompt {
            color: #CCFFFF;
            fontStyle: italic;
        }
    </fx:Style>

  <fx:Script>
    <![CDATA[
        private function resetPrompts():void {
            myTextInput.text = "";
            myTextArea.text = "";
        }
        /* Disabling the controls causes the disabledWithPrompt pseudo selector to be used
            instead of the normalWithPrompt pseudo selector. */
        private function disableControls():void {
            if (myTextInput.enabled) {
                myTextInput.enabled = false;
                myTextArea.enabled = false;
            } else {
                myTextInput.enabled = true;
                myTextArea.enabled = true;
            }
        }
      ]]>
  </fx:Script>

    <s:TextInput id="myTextInput" prompt="Enter name..."/>
    <s:TextArea id="myTextArea" prompt="Enter details..."/>

    <s:HGroup>
        <s:Button label="Reset" click="resetPrompts()"/>
        <s:Button label="Toggle Enabled/Disabled" click="disableControls()"/>
    </s:HGroup>
</s:Application>
```

Text prompt skins and transitions

The prompt text of the TextArea and TextInput controls is implemented as a Label control on the skins. The prompt text is defined in the skin as a skin part named `promptDisplay`. You can change the behavior and appearance of the prompt text by creating custom skins for those controls.

Typically, you copy the text control's skin class (for example, the TextInputSkin class) and customize it. You then apply the new skin class by pointing the `skinClass` property on the text control to your new skin.

The following example uses a custom skin called CustomTextInputSkin:

```
<?xml version="1.0"?>
<!-- sparktextcontrols/CustomPromptExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
        private function resetPrompts():void {
            myTextInput.text = "";
        }
      ]]>
  </fx:Script>

    <s:TextInput id="myTextInput" prompt="Enter name..."
        skinClass="mySkins.CustomTextInputSkin"/>
    <s:Button label="Reset" click="resetPrompts()"/>
</s:Application>
```

The following custom skin adds a transition to the prompt text when the state changes to `normalWithPrompt`. Most of the skin is directly copied from the original TextInputSkin class. The only new code is the `<s:transitions>` block near the bottom of the example. When you enter some text, and then click the Reset button, you should notice that the prompt text fades in.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- sparktextcontrols\mySkins\CustomTextInputSkin.mxml -->
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:fb="http://ns.adobe.com/flashbuilder/2009"
    alpha.disabledStates="0.5" blendMode="normal">
    <fx:Metadata>
    <![CDATA[
        [HostComponent("spark.components.TextInput")]
    ]]>
    </fx:Metadata>

    <fx:Script fb:purpose="styling">
        <![CDATA[
        import mx.core.FlexVersion;

        private var paddingChanged:Boolean;

       static private const exclusions:Array = ["background", "textDisplay", "promptDisplay",
"border"];
        static private const exclusions_4_0:Array = ["background", "textDisplay",
"promptDisplay"];

        override public function get colorizeExclusions():Array  {
            if (FlexVersion.compatibilityVersion < FlexVersion.VERSION_4_5) {
                return exclusions_4_0;
            }
            return exclusions;
        }

        static private const contentFill:Array = ["bgFill"];
        override public function get contentItems():Array {return contentFill};

        override protected function commitProperties():void {
            super.commitProperties();

            if (paddingChanged) {
                updatePadding();
                paddingChanged = false;
            }
        }
        override protected function initializationComplete():void {
            useChromeColor = true;
            super.initializationComplete();
        }

        override protected function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number):void {
            if (getStyle("borderVisible") == true) {
                border.visible = true;
                shadow.visible = true;
                background.left = background.top = background.right = background.bottom = 1;
              textDisplay.left = textDisplay.top = textDisplay.right = textDisplay.bottom = 1;
                if (promptDisplay)
                {
                    promptDisplay.setLayoutBoundsSize(unscaledWidth - 2, unscaledHeight - 2);
                    promptDisplay.setLayoutBoundsPosition(1, 1);
                }
            }
            else
            {
                border.visible = false;
                shadow.visible = false;
                background.left = background.top = background.right = background.bottom = 0;
              textDisplay.left = textDisplay.top = textDisplay.right = textDisplay.bottom = 0;
                if (promptDisplay)
                {
                    promptDisplay.setLayoutBoundsSize(unscaledWidth, unscaledHeight);
                    promptDisplay.setLayoutBoundsPosition(0, 0);
                }
            }

            borderStroke.color = getStyle("borderColor");
            borderStroke.alpha = getStyle("borderAlpha");

            super.updateDisplayList(unscaledWidth, unscaledHeight);
        }
        private function updatePadding():void {
            if (!textDisplay)
                return;

            var padding:Number;

            padding = getStyle("paddingLeft");
            if (textDisplay.getStyle("paddingLeft") != padding)
                textDisplay.setStyle("paddingLeft", padding);

            padding = getStyle("paddingTop");
            if (textDisplay.getStyle("paddingTop") != padding)
                textDisplay.setStyle("paddingTop", padding);

            padding = getStyle("paddingRight");
            if (textDisplay.getStyle("paddingRight") != padding)
                textDisplay.setStyle("paddingRight", padding);

            padding = getStyle("paddingBottom");
            if (textDisplay.getStyle("paddingBottom") != padding)
                textDisplay.setStyle("paddingBottom", padding);

            if (!promptDisplay)
                return;

            padding = getStyle("paddingLeft");
            if (promptDisplay.getStyle("paddingLeft") != padding)
                promptDisplay.setStyle("paddingLeft", padding);

            padding = getStyle("paddingTop");
            if (promptDisplay.getStyle("paddingTop") != padding)
                promptDisplay.setStyle("paddingTop", padding);

            padding = getStyle("paddingRight");
            if (promptDisplay.getStyle("paddingRight") != padding)
                promptDisplay.setStyle("paddingRight", padding);

            padding = getStyle("paddingBottom");
            if (promptDisplay.getStyle("paddingBottom") != padding)
                promptDisplay.setStyle("paddingBottom", padding);
        }

        override public function styleChanged(styleProp:String):void {
            var allStyles:Boolean = !styleProp || styleProp == "styleName";
            super.styleChanged(styleProp);

            if (allStyles || styleProp.indexOf("padding") == 0)
            {
                paddingChanged = true;
                invalidateProperties();
            }
        }
        ]]>
    </fx:Script>

    <fx:Script>
        <![CDATA[
        private static const focusExclusions:Array = ["textDisplay"];
        override public function get focusSkinExclusions():Array { return focusExclusions;};
        ]]>
    </fx:Script>

    <s:states>
        <s:State name="normal"/>
        <s:State name="disabled" stateGroups="disabledStates"/>
        <s:State name="normalWithPrompt"/>
        <s:State name="disabledWithPrompt" stateGroups="disabledStates"/>
    </s:states>
    <!-- border -->
    <s:Rect left="0" right="0" top="0" bottom="0" id="border">
        <s:stroke>
            <!--- @private -->
            <s:SolidColorStroke id="borderStroke" weight="1" />
        </s:stroke>
    </s:Rect>
    <!-- fill -->
    <s:Rect id="background" left="1" right="1" top="1" bottom="1">
        <s:fill>
            <!--- @private Defines the background fill color. -->
            <s:SolidColor id="bgFill" color="0xFFFFFF" />
        </s:fill>
    </s:Rect>

    <!-- shadow -->
    <s:Rect left="1" top="1" right="1" height="1" id="shadow">
        <s:fill>
            <s:SolidColor color="0x000000" alpha="0.12" />
        </s:fill>
    </s:Rect>
    <!-- text -->
    <s:RichEditableText id="textDisplay"
              verticalAlign="middle"
              widthInChars="10"
              left="1" right="1" top="1" bottom="1" />
    <s:Label id="promptDisplay" maxDisplayedLines="1"
                verticalAlign="middle"
                mouseEnabled="false" mouseChildren="false"
                includeIn="normalWithPrompt,disabledWithPrompt"
                includeInLayout="false"/>
    <!-- Added to custom skin to cause prompt text to fade in. -->
    <s:transitions>
        <s:Transition toState="normalWithPrompt">
            <s:Fade targets="{promptDisplay}" duration="500"/>
        </s:Transition>
    </s:transitions>

</s:SparkSkin>
```

MX text controls

MX text controls in an Adobe® Flex® application can display text, let the user enter text, or do both.

Flex defines two sets of components: Spark and MX. The Spark components are new for Flex 4 and are defined in the spark.* packages. The MX components shipped in previous releases of Flex and are defined in the mx.* packages.

Even though you can use both component sets in a single application, Adobe recommends using the Spark component set where possible. For information about the Spark text controls, see “Spark text controls” on page 734.

About MX text controls

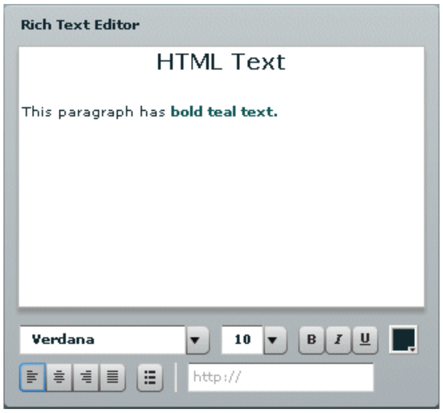
You use Flex text-based controls to display text and to let users enter text into your application. The following table lists the controls, and indicates whether the control can have multiple lines of input instead of a single line of text, and whether the control can accept user input:

MX Control	Multiline	Allows user Input
Label	No	No
TextInput	No	Yes
Text	Yes	No
TextArea	Yes	Yes
RichTextEditor	Yes	Yes

All controls except the RichTextEditor control are single components with a simple text region. For example, the following code creates an MX TextInput control in a simple form:

```
<?xml version="1.0"?>
<!-- textcontrols/FormItemLabel.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <s:Form id="myForm" width="500" backgroundColor="#909090">
     <!-- Use a FormItem to label the field. -->
     <s:FormItem label="First Name">
        <s:TextInput id="ti1" width="150"/>
     </s:FormItem>
  </s:Form>
</s:Application>
```

The RichTextEditor control is a compound control; it consists of a Panel control that contains an MX TextArea control and a ControlBar with several controls for specifying the text format and HTTP links. The following image shows a RichTextEditor control:



Rich Text Editor

HTML Text

This paragraph has **bold teal text.**

The following code produces the preceding image:

```
<?xml version="1.0"?>
<!-- textcontrols/RTECDATA.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:RichTextEditor id="rte1" title="Rich Text Editor">
     <mx:htmlText>
        <![CDATA[
           <p align='center'><b><font size='16'>HTML Text</font></b>
           This paragraph has <font color='#006666'><b>bold teal text.
           </b></font></p>
        ]]>
     </mx:htmlText>
  </mx:RichTextEditor>
</s:Application>
```

Flex text-based controls let you set and get text by using the following properties:

text Plain text without formatting information. For information on using the `text` property, see "Using the text property" on page 807.

htmlText Rich text that represents formatting by using a subset of HTML tags, and can include bulleted text and URL links. For information on using the `htmlText` property, see "Using the htmlText property" on page 810.

Both properties set the same underlying text, but you can use different formats. For example, you can do the following to set, modify, and get text:

- You can set formatted text by using the `htmlText` property, and get it back as a plain text string by using the `text` property.
- You can set formatted text in user-editable text controls (MX TextInput, TextArea, RichTextEditor) by setting the text string with the `text` property and formatting a section of this text by using the TextRange class. If you get the text back by using the `htmlText` property, the property string includes HTML tags for the formatting. For more information on using the TextRange class, see "Selecting and modifying text" on page 820.

Using the text property

You can use the text property to specify the text string that appears in a text control or to get the text in the control as a plain text String. When you set this property, any HTML tags in the text string appear in the control as literal text.

You cannot specify text formatting when you set the `text` property, but you can format the text in the control. You can use the text control styles to format all of the text in the control, and you can use the TextRange class to format ranges of text**.** **(**For more information on using the TextRange class, see "Selecting and modifying text" on page 820.)

The following code line uses a `text` property to specify label text:

```
<mx:Label text="This is a simple text label"/>
```

The way you specify special characters, including quotation marks, greater than and less than signs, and apostrophes, depends on whether you use them in MXML tags or in ActionScript. It also depends on whether you specify the text directly or wrap the text in a CDATA section.

***Note:** If you specify the value of the `text` property by using a string directly in MXML, Flex collapses white space characters. If you specify the value of the `text` property in ActionScript, Flex does not collapse white space characters.*

Specifying special characters in the text property

The following rules specify how to include special characters in the `text` property of a text control MXML tag, either in a property assignment, such as `text="the text"`, or in the body of an `<mx:text>` subtag.

In standard text The following rules determine how you use special characters if you do not use a CDATA section:

- To use the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), insert the XML character entity equivalents of `&lt;`, `&gt;`, and `&amp;`, respectively. You can also use `&quot;` and `&apos;` for double-quotation marks (") and single-quotation marks ('), and you can use numeric character references, such as &#165 for the Yen mark (¥). Do not use any other named character entities; Flex treats them as literal text.
- You cannot use the character that encloses the property text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence `\"` for any double-quotation marks in the string. If you surround the string in single-quotation marks (') use the escape sequence `\'` for any single-quotation marks in the string. You *can* use single-quotation marks inside a string that is surrounded in double-quotation marks, and double-quotation marks inside a string that is surrounded in single-quotation marks.
- Flex text controls ignore escape characters such as \t or \n in the `text` property. They ignore or convert to spaces, tabs and line breaks, depending on whether you are specifying a property assignment or an `<mx:text>` subtag. To include line breaks, put the text in a CDATA section. In the Text control `text="`*string*`"` attribute specifications, you can also specify them as numeric character entities, such as &#013; for a Return character or &#009; for a Tab character, but you cannot do this in an `<mx:text>` subtag.

The following code example uses the `text` property with standard text:

```
<?xml version="1.0"?>
<!-- textcontrols/StandardText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="400">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <mx:Text width="400"
    text="This string contains a less than, &lt;,
    greater than, &gt;, ampersand, &amp;, apostrophe, ', and
    quotation mark &quot;."/>
  <mx:Text width="400"
    text='This string contains a less than, &lt;,
    greater than, &gt;, ampersand, &amp;, apostrophe, &apos;, and
    quotation mark, ".'/>
  <mx:Text width="400">
     <mx:text>
        This string contains a less than, &lt;, greater than,
        &gt;, ampersand, &amp;, apostrophe, ', and quotation mark, ".
     </mx:text>
  </mx:Text>
</s:Application>
```

The resulting application contains three almost identical text controls, each with the following text. The first two controls, however, convert any tabs in the text to spaces.

```
This string contains a less than, <, greater than, >, ampersand, &,apostrophe, ', and quotation
mark, ".
```

In a CDATA section If you wrap the text string in the `CDATA` tag, the following rules apply:

- You cannot use a CDATA section in a property assignment statement in the text control opening tag; you must define the property in an `<mx:text>` child tag.
- Text inside the CDATA section appears as it is entered, including white space characters. Use literal characters, such as " or < for special characters, and use standard return and tab characters. Character entities, such as &gt;, and backslash-style escape characters, such as \n, appear as literal text.

The following code example follows these CDATA section rules. The second and third lines of text in the `<mx:text>` tag are not indented because any leading tab or space characters would appear in the displayed text.

```
<?xml version="1.0"?>
<!-- textcontrols/TextCDATA.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="500">

  <mx:Text width="100%">
     <mx:text>
        <![CDATA[This string contains a less than, <, greater than, >,
            ampersand, &, apostrophe, ', return,
            tab. and quotation mark, ".]]>
     </mx:text>
  </mx:Text>
</s:Application>
```

The displayed text appears on three lines, as follows:

```
This string contains a less than, <, greater than, >,
ampersand, &, apostrophe, ', return,
tab. and quotation mark, ".
```

Specifying special characters in ActionScript

The following rules specify how to include special characters in a text control when you specify the control's `text` property value in ActionScript; for example, in an initialization function, or when assigning a string value to a variable that you use to populate the property:

- You cannot use the character that encloses the text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence \" for any double-quotation marks in the string. If you surround the string in single-quotation marks ('), use the escape sequence \' for any single-quotation marks in the string.
- Use backslash escape characters for special characters, including \t for the tab character, and \n or \r for a return/line feed character combination. You can use the escape character \" for the double-quotation mark and \' for the single-quotation mark.
- In standard text, but not in CDATA sections, you can use the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), by inserting the XML character entity equivalents of `&lt;`, `&gt;`, and `&amp;`, respectively. You can also use `&quot;` and `&apos;` for double-quotation marks ("), and single-quotation marks ('), and you can use numeric character references, such as `&#165;` for the Yen mark (¥). Do not use any other named character entities; Flex treats them as literal text.
- In CDATA sections only, do not use character entities or references, such as `&lt;` or &#165; because Flex treats them as literal text. Instead, use the actual character, such as <.

The following example uses an initialization function to set the `text` property to a string that contains these characters:

```
<?xml version="1.0"?>
<!-- textcontrols/InitText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initText()">

  <fx:Script>
     public function initText():void {
        //The following is on one line.
        myText.text="This string contains a return, \n, tab, \t, and quotation mark, \". " +
            "This string also contains less than, &lt;, greater than, &gt;, " +
            "ampersand, &amp;, and apostrophe, ', characters.";
     }
  </fx:Script>
  <mx:Text width="450" id="myText"/>
</s:Application>
```

The following example uses an `<fx:Script>` tag with a variable in a CDATA section to set the `text` property:

```
<?xml version="1.0"?>
<!-- textcontrols/VarText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            [Bindable]
            // The following is on one line.
          public var myText:String ="This string contains a return, \n, tab, \t, and quotation
mark, \". This string also contains less than, <, greater than, <, ampersand, <;, and
apostrophe, ', characters.";
        ]]>
    </fx:Script>
    <mx:Text width="450" text="{myText}"/>

</s:Application>
```

The displayed text for each example appears on three lines. The first line ends at the return specified by the \n character. The remaining text wraps onto a third line because it is too long to fit on a single line. (Note: Although the tab character may be noticeable in the following output, it is included in the right location.)

```
This string contains a return,
, tab, , and quotation mark, ". This string also contains less than, <,
greater than, >, ampersand, &, and apostrophe, ', characters.
```

Using the htmlText property

You use the `htmlText` property to set or get an HTML-formatted text string. You can also use one tag that is not part of standard HTML, the `textFormat` tag. For details of supported tags and attributes, see “Using tags in HTML text” on page 813.

You can also specify text formatting by using Flex styles. You can set a base style, such as the font characteristics or the text weight, by using a style, and override the base style in sections of your text by using tags, such as the `<font>` tag. In the following example, the `<mx:Text>` tag styles specify blue, italic, 14 point text, and the `<mx:htmlText>` tag includes HTML tags that override the color and point size.

```
<?xml version="1.0"?>
<!-- textcontrols/HTMLTags.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <mx:Text width="100%" color="blue" fontStyle="italic" fontSize="14">
        <mx:htmlText>
            <![CDATA[
                This is 14 point blue italic text.<br>
                <b><font color="#000000" size="10">This text is 10 point black, italic, and 
bold.</font></b>
            ]]>
        </mx:htmlText>
    </mx:Text>
</s:Application>
```

Specifying HTML tags and text

To prevent the Flex compiler from generating errors when it encounters HTML tags in the text, use one of the following techniques:

- Wrap your text in a `CDATA` tag.
- Specify HTML markup by using the `&lt;`, `&gt;`, and `&amp;` character entities in place of the left angle bracket (<), right angle bracket (>), and ampersand (&) HTML delimiters.

Adobe recommends using CDATA sections for all but simple HTML markup, because the character entity technique has significant limitations:

- Extensive HTML markup can be cumbersome to write and difficult to read.
- You must use a complex escape sequence to include the less than and ampersand characters in your text.

For example, to display the following string:

A less than character < and **bold text**.

without using a CDATA section, you must use the following text:

```
A less than character &amp;c#060; and &lt;b&gtbold text&lt;/b&gt.
```

In a CDATA section, you use the following text:

```
A less than character &lt; and <b>bold text</b>.
```

Specifying HTML text

When you specify HTML text for a text control, the following rules apply:

- You cannot use a CDATA section directly in an inline `htmlText` property in an `<mx:Text>` tag. You must put the text in an `<mx:htmlText>` subtag, or in ActionScript code.
- Flex collapses consecutive white space characters, including return, space, and tab characters, in text that you specify in MXML property assignments or ActionScript outside of a CDATA section.

- If you specify the text in a CDATA section, you can use the text control's `condenseWhite` property to control whether Flex collapses white space. By default, the `condenseWhite` property is `false`, and Flex does not collapse white space.
- Use HTML <p> and <br> tags for breaks and paragraphs. In ActionScript CDATA sections you can also use \n escape characters.
- If your HTML text string is surrounded by single- or double-quotation marks because it is in an assignment statement (in other words, if it is not in an `<mx:htmlText>` tag), you must escape any uses of that quotation character in the string:
    - If you use double-quotation marks for the assignment delimiters, use &quot; for the double-quotation mark (") character in your HTML. In ActionScript, you can also use the escape sequence \".

      ***Note:** You do not need to escape double-quotation marks if you're loading text from an external file; it is only necessary if you're assigning a string of text in ActionScript.*
    - If you use single-quotation marks for the assignment delimiters, use &apos; for the single-quotation mark character (') in your HTML. In ActionScript, you can also use the escape sequence \'.
- When you enter HTML-formatted text, you must include attributes of HTML tags in double- or single-quotation marks. Attribute values without quotation marks can produce unexpected results, such as improper rendering of text. You must follow the escaping rules for quotation marks within quotation marks, as described in "Escaping special characters in HTML text" on page 813.

The following example shows some simple HTML formatted text, using MXML and ActionScript to specify the text:

```
<?xml version="1.0"?>
<!-- textcontrols/HTMLFormattedText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="500">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Script><![CDATA[
     //The following is on one line.
     [Bindable]
     public var myHtmlText:String="This string contains <b>less than </b>, &lt;, <b>greater
than</b>, &gt;, <b>ampersand</b>, &amp;, and <b>double quotation mark</b>, &quot;,
characters.";
  ]]></fx:Script>
  <mx:Text id="htmltext2" width="450" htmlText="{myHtmlText}" />
  <mx:Text width="450">
     <mx:htmlText>
        <!-- The following is on one line. Line breaks would appear in the output. -->
        <![CDATA[
           This string contains <b>less than</b>, &lt;, <b>greater than </b>, &gt;,
<b>ampersand</b>, &amp;, and <b>double quotation mark</b>,&quot;, characters.
        ]]>
     </mx:htmlText>
  </mx:Text>
</s:Application>
```

Escaping special characters in HTML text

The rules for escaping special characters in HTML text differ between CDATA sections and standard text.

In CDATA sections When you specify the `htmlText` string, the following rules apply:

- In ActionScript, but not in an `<mx:htmlText>` tag, you can use standard backslash escape sequences for special characters, such as \t for tab and \n for a newline character. You can also use the backslash character to escape many special characters, such as \\xd5 and \" for single- and double-quotation marks. You cannot use the combination \<, and a backslash before a return character has no effect on displayed text; it allows you to break the assignment statement across multiple text lines.
- In both ActionScript and the `<mx:htmlText>` tag, you can use HTML tags and numeric character entities; for example in place of \n, you can use a `<br>` tag.
- To include a left angle bracket (<), right angle bracket (>), or ampersand (&) character in displayed text, use the corresponding character entities: `&lt;`, `&gt;`, and `&amp;, respectively`. You can also use the `&quot;` and `&apos;` entities for single- and double-quotation marks. These are the only named character entities that Adobe® Flash® Player and Adobe® AIR™ recognize. They recognize numeric entities, such as `&#165;` for the Yen mark (¥),;however, they do not recognize the corresponding character entity, `&yen;`.

  The following code example uses the `htmlText` property to display formatted text:

```
<?xml version="1.0"?>
<!-- textcontrols/HTMLTags2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="500">
  <mx:Text width="100%">
     <mx:htmlText><![CDATA[<p>This string contains a <b>less than</b>, &lt;.
        </p><p>This text is in a new paragraph.<br>This is a new line.</p>]]>
     </mx:htmlText>
  </mx:Text>
</s:Application>
```

In standard text The following rules apply:

- You must use character entities, as described in "Using the htmlText property" on page 810, to use the left angle bracket (<), right angle bracket (>), or ampersand (&) character in HTML; for example, when you open a tag or start a character entity.
- You must use the `&amp;` named entity combined with an HTML numeric character entity to display the less than character (use `&amp;#060;`) and ampersand character (use `&amp;#038;`). You can use the standard character entities, `&gt;`, `&quot;`, and `&apos;`, for the greater than, double-quotation mark and single-quotation mark characters, respectively. For all other character entities, use numeric entity combinations, such as `&amp;#165;`, for the Yen mark (¥).
- In ActionScript, but not in an `<mx:htmlText>` tag or inline `htmlText` property, you can use a backslash character to escape special characters, including the tab, newline, and quotation mark characters (but not the ampersand). In all cases, you can use (properly escaped) HTML tags and numeric character entities; for example in place of \n, you can use a `&lt;br&gt;` tag or `&amp;#013;` entity.

Using tags in HTML text

When you use the `htmlText` property, you use a subset of HTML that is supported by Flash Player and AIR, which support the following tags:

Anchor tag (<a>)

The anchor `<a>` tag creates a hyperlink and supports the following attributes:

href Specifies the URL of the page to load in the browser. The URL can be absolute or relative to the location of the SWF file that is loading the page.

target Specifies the name of the target window to load the page into.

For example, the following HTML snippet creates the link “Go Home” to the Adobe Web site.

```
<a href='http://www.adobe.com' target='_blank'>Go Home</a>
```

The `<a>` tag does *not* make the link text blue or underline the text. You must apply formatting tags to change the text format. You can do this with the `<font color="`*`color`*`">` tag and the `<u>` tag.

You can also define `a:link`, `a:hover`, and `a:active` styles for anchor tags by using the StyleSheet class, if the component supports the `styleSheet` property. This property is defined on MX TextArea and TextField controls. The following example shows how to use it on text inside an MX TextArea control:

```
<?xml version="1.0"?>
<!-- textcontrols/StyleSheetExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            import flash.text.StyleSheet;
            private function initApp():void {
                    var ss:StyleSheet = new StyleSheet;
                    // Define an object for the "hover" state of the "a" tag.
                    var hoverStyles:Object = new Object;
                    hoverStyles.textDecoration = "underline";
                    hoverStyles.color = "#FF00CC";
                    // Define an object for the non-hover state of the "a" tag.
                    var linkStyles:Object = new Object;
                    linkStyles.color = "#FF00CC";
                    // Apply the newly defined styles.
                    ss.setStyle("a:hover", hoverStyles);
                    ss.setStyle("a", linkStyles);
                    // Apply the StyleSheet to the TextArea control.
                    myTA.styleSheet = ss;
                }
            ]]>
    </fx:Script>

    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|TextArea {
            fontFamily:Courier;
            linkcolor:#CC3300;
        }
    </fx:Style>

    <mx:TextArea id="myTA" height="100" width="200">
        <mx:htmlText>
            <![CDATA[<a href="http://www.adobe.com">This</a> is a link.]]>
        </mx:htmlText>
    </mx:TextArea>
</s:Application>
```

The MX Label, Text, and TextArea controls can dispatch a `link` event when the user selects a hyperlink in the `htmlText` property. To generate the `link` event, prefix the hyperlink destination with `event:`, as the following example shows:

```
<?xml version="1.0"?>
<!-- textcontrols/LabelControlLinkEvent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import flash.events.TextEvent;
            public function linkHandler(event:TextEvent):void {
                myTA.text="The link was clicked.";

                // Open the link in a new browser window.
                navigateToURL(new URLRequest(event.text), '_blank')
            }
        ]]>
    </fx:Script>

    <mx:Label selectable="true" link="linkHandler(event);">
        <mx:htmlText>
            <![CDATA[This link lets you <a href='event:http://www.adobe.com'>Navigate to 
Adobe.com.</a>]]>
        </mx:htmlText>
    </mx:Label>
    <mx:TextArea id="myTA"/>

</s:Application>
```

The MX Label control must have the `selectable` property set to `true` to generate the `link` event.

When you use the `link` event, the event is generated and the text following `event:` in the hyperlink destination is included in the `text` property of the event object. However, the hyperlink is not automatically executed; you must execute the hyperlink from within your event handler. This allows you to modify the hyperlink, or even prohibit it from occurring, in your application.

Bold tag (<b>)

The bold `<b>` tag renders text as bold. If you use embedded fonts, a boldface font must be available for the font or no text appears. If you use fonts that you expect to reside on the local system of your users, their system may approximate a boldface font if none exists, or it may substitute the normal font face instead of boldface. In either case, the text inside the bold tags will appear.

The following snippet applies boldface to the word *bold*:

```
This word is <b>bold</b>.
```

You cannot use the `</b>` end tag to override bold formatting that you set for all text in a control by using the `fontWeight` style.

Break tag (<br>)

The break `<br>` tag creates a line break in the text. This tag has no effect in MX Label or MX TextInput controls.

The following snippet starts a new line after the word *line*:

```
The next sentence is on a new line.<br>Hello there.
```

Font tag (<font>)

The `<font>` tag specifies the following font characteristics: color, face, and size.

The font tag supports the following attributes:

color Specifies the text color. You must use hexadecimal (`#FFFFFF`) color values. Other formats are not supported.

face Specifies the name of the font to use. You can also specify a list of comma-separated font names, in which case Flash Player and AIR choose the first available font. If the specified font is not installed on the playback system, or isn't embedded in the SWF file, Flash Player and AIR choose a substitute font. The following example shows how to set the font face.

size Specifies the size of the font in points. You can also use relative sizes (for example, +2 or -4).

The following example shows the use of the `<font>tag:`

```
<?xml version="1.0"?>
<!-- textcontrols/FontTag.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

  <mx:TextArea height="100" width="250">
     <mx:htmlText>
        <![CDATA[
           You can vary the <font size='20'>font size</font>,<br><font
color="#0000FF">color</font>,<br><font face="CourierNew, Courier, Typewriter">face</font>,
or<br><font size="18" color="#FF00FF"face="Times, Times New Roman, _serif">any combination of
the three.</font>
        ]]>
     </mx:htmlText>
  </mx:TextArea>
</s:Application>
```

Image tag (<img>)

***Note:** The `<img>` tag is not fully supported, and might not work in some cases.*

The image `<img>` tag lets you embed external JPEG, GIF, PNG, and SWF files inside text fields. Text automatically flows around images you embed in text fields. This tag is supported only in dynamic and input text fields that are multiline and wrap their text.

By default, Flash displays media embedded in a text field at full size. To specify dimensions for embedded media, use the `<img>` tag's `height` and `width` attributes.

In general, an image embedded in a text field appears on the line following the `<img>` tag. However, when the `<img>` tag is the first character in the text field, the image appears on the first line of the text field.

The `<img>` tag has one required attribute, `src`, which specifies the path to an image file. All other attributes are optional.

The `<img>` tag supports the following attributes:

src Specifies the URL to a GIF, JPEG, PNG, or SWF file. This attribute is required; all other attributes are optional. External files are not displayed until they have downloaded completely.

align Specifies the horizontal alignment of the embedded image within the text field. Valid values are `left` and `right`. The default value is `left`.

height Specifies the height of the image, in pixels.

hspace Specifies the amount of horizontal space that surrounds the image where no text appears. The default value is 8.

id Specifies the identifier for the imported image. This is useful if you want to control the embedded content with ActionScript.

vspace Specifies the amount of vertical space that surrounds the image where no text.

width Specifies the width of the image, in pixels. The default value is 8.

The following example shows the use of the `<img>` tag and how text can flow around the image:

```
<?xml version="1.0"?>
<!-- textcontrols/ImgTag.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="300" height="300">
  <mx:Text height="100%" width="100%">
     <mx:htmlText>
        <![CDATA[
           <p>You can include an image in your HTML text with the &lt;img&gt; tag.</p>
          <p><img src='assets/butterfly.gif' width='30' height='30' align='left' hspace='10'
vspace='10'>
           Here is text that follows the image. I'm extending the text by lengthening this
sentence until it's long enough to show wrapping around the bottom of the image.</p>
        ]]>
     </mx:htmlText>
  </mx:Text>
</s:Application>
```

Making hyperlinks out of embedded images

To make a hyperlink out of an embedded image, enclose the `<img>` tag in an `<a>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- textcontrols/ImgTagWithHyperlink.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:TextArea width="100%" height="100%">
     <mx:htmlText>
        <![CDATA[
           <a href='http://www.adobe.com'><img src='assets/butterfly.gif'/></a>
           Click the image to go to the Adobe home page.
        ]]>
     </mx:htmlText>
  </mx:TextArea>
</s:Application>
```

When the user moves the mouse pointer over an image that is enclosed by `<a>` tags, the mouse pointer does not change automatically to a hand icon, as with standard hyperlinks. To display a hand icon, specify `buttonMode="true"` for the MX TextArea (or Text) control. Interactivity, such as mouse clicks and key presses, do not register in SWF files that are enclosed by `<a>` tags.

Italic tag (<i>)

The italic `<i>` tag displays the tagged text in italic font. If you're using embedded fonts, an italic font must be available or no text appears. If you use fonts that you expect to reside on the local system of your users, their system may approximate an italic font if none exists, or it may substitute the normal font face instead of italic. In either case, the text inside the italic tags appears.

The following snippet applies italic font to the word *italic*:

```
The next word is in <i>italic</i>.
```

You cannot use the `</i>` end tag to override italic formatting that you set for all text in a control by using the `fontStyle` style.

List item tag (<li>)

The list item `<li>` tag ensures that the text that it encloses starts on a new line with a bullet in front of it. You cannot use it for any other type of HTML list item. The ending `</li>` tag ensures a line break (but `</li><li>` generates a single line break). Unlike in HTML, you do not surround `<li>` tags in `<ul>` tags. For example, the following Flex code generates a bulleted list with two items:

```
<?xml version="1.0"?>
<!-- textcontrols/BulletedListExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:Text>
     <mx:htmlText >
        <![CDATA[
           <p>This is a bulleted list:<li>First Item</li><li>Second Item</li></p>
        ]]>
     </mx:htmlText>
  </mx:Text>
</s:Application>
```

***Note:** The `<li>` tag does not work properly with MX Label controls. With MX TextInput controls, it must be put before the first character in the text.*

Paragraph tag (<p>)

The paragraph `<p>` tag creates a new paragraph. The opening `<p>` tag does *not* force a line break, but the closing `</p>` tag does. Unlike in HTML, the `<p>` tag does not force a double space between paragraphs; the spacing is the same as that generated by the `<br>` tag.

The `<p>` tag supports the following attribute:

align Specifies alignment of text in the paragraph; valid values are `left`, `right`, `center`, and `justify`.

The following snippet generates two centered paragraphs:

```
<p align="center">This is a first centered paragraph</p>
<p align="center">This is a second centered paragraph</p>
```

Text format tag (<textformat>)

The text format `<textformat>` tag lets you use a subset of paragraph formatting properties of the TextFormat class in HTML text fields, including line leading, indentation, margins, and tab stops. You can combine text format tags with the built-in HTML tags. The text format tag supports the following attributes:

blockindent Specifies the indentation, in points, from the left margin to the text in the `<textformat>` tag body.

indent Specifies the indentation, in points, from the left margin or the block indent, if any, to the first character in the `<textformat>` tag body.

leading Specifies the amount of leading (vertical space) between lines.

leftmargin Specifies the left margin of the paragraph, in points.

rightmargin Specifies the right margin of the paragraph, in points.

tabstops Specifies custom tab stops as an array of nonnegative integers.

Underline tag (<u>)

The underline `<u>` tag underlines the tagged text.

The following snippet underlines the word *underlined*:

```
The next word is <u>underlined</u>.
```

You cannot use the `</u>` end tag to override underlining that you set for all text in a control by using the `textDecoration` style.

Selecting and modifying text

You can select and modify text in the MX TextArea, MX TextInput, and RichTextEditor controls. To change a MX Label or Text control's text, assign a new value to the control's `text` or `HTMLtext` property. For more information on the `HTMLText` property, see "Using the htmlText property" on page 810.

Selecting text

The Flex editable controls provide properties and methods to select text regions and get selections. You can modify the contents of the selection as described in "Modifying text" on page 821.

Creating a selection

The MX TextInput and TextArea controls, including the RichTextEditor control's TextArea subcontrol, provide the following text selection properties and method:

- `setSelection()` method selects a range of text. You specify the zero-based indexes of the start character and the position immediately *after* the last character in the text.
- `selectionBeginIndex` and `selectionEndIndex` set or return the zero-based location in the text of the start and position immediately *after* the end of a selection.

To select the first 10 characters of the myTextArea TextArea control, for example, use the following method:

```
myTextArea.setSelection(0, 10);
```

To change the last character of this selection to be the twenty-fifth character in the MX TextArea control, use the following statement:

```
myTextArea.endIndex=25;
```

To select text in a RichTextEditor control, use the control's TextArea subcontrol, which you access by using the `textArea` id. To select the first 10 characters in the myRTE RichTextEditor control, for example, use the following code:

```
myRTE.textArea.setSelection(0, 10);
```

Getting a selection

You get a text control's selection by getting a TextRange object with the selected text. You can then use the TextRange object to modify the selected text, as described in "Modifying text" on page 821. The technique you use to get the selection depends on the control type.

Get the selection in an MX TextArea or MX TextInput control

Use the TextRange class constructor to get a TextRange object with the currently selected text in an MX TextArea or MX TextInput control. For example, to get the current selection of the myTextArea control, use the following line:

```
var mySelectedTextRange:TextRange = new TextRange(myTextArea, true);
```

The second parameter, `true`, tells the constructor to return a TextRange object with the selected text.

Get the selection in a RichTextEditor control

Use the `selection` read-only property of the RichTextEditor to get a TextRange object with the currently selected text in its MX TextArea subcontrol. You can use the TextRange object to modify the selected text, as described in Modifying text. For example, to get the current selection of the MyRTE RichTextEditor control, us the following line:

```
public var mySelectedTextRange:TextRange = myRTE.selection;
```

Modifying text

You use the TextRange class to modify the text in an MX TextArea, MX TextInput, or RichTextEditor control. This class lets you affect the following text characteristics:

- `text` or `htmltext` property contents
- text color, decoration (underlining), and alignment
- font family, size, style (italics), and weight (bold)
- URL of an HTML <a> link

Getting a TextRange object

To get a TextRange object you use the following techniques:

- Get a TextRange object that contains the current text selection.
- Create a TextRange object that contains a specific range of text.

To create a TextRange object with a specific range of text, use a TextRange constructor with the following format:

```
new TextRange(control, modifiesSelection, beginIndex, endIndex)
```

Specify the control that contains the text, whether the TextRange object corresponds to a selection (that is, represents and modifies selected text), and the zero-based indexes in the text of the first and last character of the range. As a general rule, do not use the TextRange constructor to set a selection; use the `setSelection()` method, as described in “Selecting text” on page 820. For this reason, the second parameter should always be `false` when you specify the begin and end indexes.

To get a TextRange object with the fifth through twenty-fifth characters of an MX TextArea control named myTextArea, for example, use the following line:

```
var myTARange:TextRange = new TextRange(myTextArea, false, 4, 25);
```

Changing text

After you get a TextRange object, use its properties to modify the text in the range. The changes you make to the TextRange appear in the text control.

You can get or set the text in a TextRange object as HTML text or as a plain text, independent of any property that you might have used to initially set the text. If you created an MX TextArea control, for example, and set its `text` property, you can use the TextRange `htmlText` property to get and change the text. The following example shows this usage, and shows using the TextRange class to access a range of text and change its properties. It also shows using String properties and methods to get text indexes.

```
<?xml version="1.0"?>
<!-- textcontrols/TextRangeExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
     import mx.controls.textClasses.TextRange
    private function resetText():void {
        ta1.text = "This is a test of the emergency broadcast system. It is only a test.";
    }
     public function alterText():void {
        // Create a TextRange object starting with "the" and ending at the
        // first period. Replace it with new formatted HTML text.
        var tr1:TextRange = new TextRange(
            ta1, false, ta1.text.indexOf("the", 0), ta1.text.indexOf(".", 0)
        );
        tr1.htmlText="<i>italic HTML text</i>"
        // Create a TextRange object with the remaining text.
        // Select the text and change its formatting.
        var tr2:TextRange = new TextRange(
            ta1, true, ta1.text.indexOf("It", 0), ta1.text.length-1
        );
        tr2.color=0xFF00FF;
        tr2.fontSize=18;
        tr2.fontStyle = "italic"; // any other value turns italic off
        tr2.fontWeight = "bold"; // any other value turns bold off
        ta1.setSelection(0, 0);
     }
  ]]></fx:Script>
  <mx:TextArea id="ta1" fontSize="12" fontWeight="bold" width="100%" height="100">
     <mx:text>
        This is a test of the emergency broadcast system. It is only a test.
     </mx:text>
  </mx:TextArea>
  <s:HGroup>
    <mx:Button label="Alter Text" click="alterText();"/>
    <mx:Button label="Reset" click="resetText();"/>
  </s:HGroup>
</s:Application>
```

Example: Changing selected text in a RichTextEditor control

The following example shows how you can use the `selectedText` property of theRichTextEditor control to get a TextRange when a user selects some text, and use TextRange properties to get and change the characteristics of the selected text. To use the example, select a range of text with your mouse. When you release the mouse button, the string "This is replacement text. ", formatted in fuchsia Courier 20-point font replaces the selection and the text area reports on the original and replacement text.

```
<?xml version="1.0"?>
<!-- textcontrols/TextRangeSelectedText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600" height="500">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Script><![CDATA[
     import mx.controls.textClasses.TextRange;
     //The following text must be on a single line.
     [Bindable]
     public var htmlData:String="<textformat leading='2'><p align='center'><b><font 
size='20'>HTML Formatted Text</font></b></p></textformat><br><textformat leading='2'><p 
align='left'><font face='_sans' size='12' color='#000000'>This paragraph contains <b>bold</b>, 
<i>italic</i>, <u>underlined</u>, and <b><i><u>bold italic underlined </u></i></b>text.
</font></p></textformat><br><p><u><font face='arial' size='14' color='#ff0000'>This a red 
underlined 14-point arial font with no alignment set.</font></u></p><p align='right'><font 
face='verdana' size='12' color='#006666'><b>This a teal bold 12-pt. Verdana font with alignment 
set to right.</b></font></p>";
     public function changeSelectionText():void {
        //Get a TextRange with the selected text and find its length.
        var sel:TextRange = rte1.selection;
        var selLength:int = sel.endIndex - sel.beginIndex;
        //Do the following only if the user made a selection.
        if (selLength) {
           //Display the selection size and font color, size, and family.
           t1.text="Number of characters selected: " + String(selLength);
           t1.text+="\n\nOriginal Font Family: " + sel.fontFamily;
           t1.text+="\nOriginal Font Size: " + sel.fontSize;
           t1.text+="\nOriginal Font Color: " + sel.color;
           //Change font color, size, and family and replace selected text.
           sel.text="This is replacement text. "
           sel.color="fuchsia";
           sel.fontSize=20;
           sel.fontFamily="courier"
           //Show the new font color, size, and family.
```

```
            t1.text+="\n\nNew text length: " + String(sel.endIndex - sel.beginIndex);
            t1.text+="\nNew Font Family: " + sel.fontFamily;
            t1.text+="\nNew Font Size: " + sel.fontSize;
            t1.text+="\nNew Font Color: " + sel.color;
         }
      }
   ]]></fx:Script>
   <!-- The text area. When you release the mouse after selecting text,
      it calls the func1 function. -->
   <mx:RichTextEditor id="rte1"
     htmlText="{htmlData}"
     width="100%" height="100%"
     mouseUp="changeSelectionText()"/>

   <mx:TextArea id="t1"
     editable="false"
     fontSize="12"
     fontWeight="bold"
     width="300" height="180"/>
</s:Application>
```

MX Label control

The Label control is part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead. For more information about using Spark text controls, see “Spark text controls” on page 734.

The MX Label control is a noneditable single-line text label. It has the following characteristics:

- The user cannot change the text, but the application can modify it.
- You can specify text formatting by using styles or HTML text.
- You can control the alignment and sizing.
- The control is transparent and does not have a `backgroundColor` property, so the background of the component’s container shows through.
- The control has no borders, so the label appears as text written directly on its background.
- The control cannot take the focus.

To create a multiline, noneditable text field, use a Text control. For more information, see “MX Text control” on page 826. To create user-editable text fields, use MX TextInput or MX TextArea controls. For more information, see “MX TextInput control” on page 825 and “MX TextArea control” on page 827.

The following image shows an MX Label control:

For the code used to create this sample, see Creating an MX Label control.

Creating an MX Label control

You define an MX Label control in MXML by using the `<mx:Label>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/LabelControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="150" height="80">
  <mx:Label text="Label1"/>
</s:Application>
```

You use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information on using these properties, see "Using the text property" on page 807 and "Using the htmlText property" on page 810.

Sizing an MX Label control

If you do not specify a width, the MX Label control automatically resizes when you change the value of the `text` or `htmlText` property.

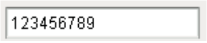
If you explicitly size an MX Label control so that it is not large enough to accommodate its text, the text is truncated and terminated by an ellipsis (...). The full text displays as a tooltip when you move the mouse over the MX Label control. If you also set a tooltip by using the `tooltip` property, the tooltip is displayed rather than the text.

MX TextInput control

The TextInput control is part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead. For more information about using Spark text controls, see "Spark text controls" on page 734.

The MX TextInput control is a single-line text field that is optionally editable. The MX TextInput control supports the HTML rendering capabilities of Adobe Flash Player.

The following image shows an MX TextInput control:



123456789

To create a multiline, editable text field, use an MX TextArea control. For more information, see "MX TextArea control" on page 827. To create noneditable text fields, use MX Label and Text controls. For more information, see "MX Label control" on page 824 and "MX Text control" on page 826.

The MX TextInput control does not include a label, but you can add one by using an MX Label control or by nesting the MX TextInput control in a FormItem container in a Form layout container, as shown in the example in "About MX text controls" on page 805. MX TextInput controls dispatch `change`, `textInput`, and `enter` events.

If you disable an MX TextInput control, it displays its contents in a different color, represented by the `disabledColor` style. You can set an MX TextInput control's `editable` property to `false` to prevent editing of the text. You can set an MX TextInput control's `displayAsPassword` property to conceal the input text by displaying characters as asterisks.

Creating an MX TextInput control

You define an MX TextInput control in MXML by using the `<mx:TextInput>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/TextInputControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:TextInput id="text1" width="100"/>
</s:Application>
```

Just as you can for the MX Label control, you use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see "Using the text property" on page 807 and "Using the htmlText property" on page 810.

Sizing an MX TextInput control

If you do not specify a width, the MX TextInput control automatically resizes when you change the value of the `text` or `htmlText` property. It does not resize in response to typed user input.

Binding to an MX TextInput control

In some cases, you might want to bind a variable to the text property of an MX TextInput control so that the control represents a variable value, as the following example shows:

```
<?xml version="1.0"?>
<!-- textcontrols/BoundTextInputControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script><![CDATA[
     [Bindable]
     public var myProp:String="This is the initial String myProp.";
  ]]></fx:Script>
  <mx:TextInput text="{myProp}" width="250"/>
</s:Application>
```

In this example, the MX TextInput control displays the value of the myProp variable. Remember that you must use the `[Bindable]` metadata tag if the variable changes value and the control must track the changed values; also, the compiler generates warnings if you do not use this metadata tag.

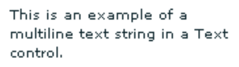
MX Text control

The MX Text control displays multiline, noneditable text. The control has the following characteristics:

- The user cannot change the text, but the application can modify it.
- The control does not support scroll bars. If the text exceeds the control size, users can use keys to scroll the text.
- The control is transparent so that the background of the component's container shows through.
- The control has no borders, so the label appears as text written directly on its background.
- The control supports HTML text and a variety of text and font styles.
- The text always word-wraps at the control boundaries, and is always aligned to the top of the control.

To create a single-line, noneditable text field, use the MX Label control. For more information, see "MX Label control" on page 824. To create user-editable text fields, use the MX TextInput or TextArea controls. For more information, see "MX TextInput control" on page 825 and "MX TextArea control" on page 827.

The following image shows an example of a Text control with a width of 175 pixels:



This is an example of a
multiline text string in a Text
control.

Creating a Text control

You define a Text control in MXML by using the `<mx:Text>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/TextControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Text width="175"
        text="This is an example of a multiline text string in a Text control."/>
</s:Application>
```

You use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see "Using the text property" on page 807 and "Using the htmlText property" on page 810.

This control does not support a `backgroundColor` property; its background is always the background of the control's container.

Sizing a Text control

Flex sizes the Text control as follows:

- If you specify a pixel value for both the `height` and `width` properties, any text that exceeds the size of the control is clipped at the border.
- If you specify an explicit pixel width, but no height, Flex wraps the text to fit the width and calculates the height to fit the required number of lines.
- If you specify a percentage-based width and no height, Flex does *not* wrap the text, and the height equals the number of lines as determined by the number of Return characters.
- If you specify only a height and no width, the height value does not affect the width calculation, and Flex sizes the control to fit the width of the maximum line.

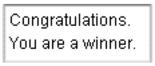
As a general rule, if you have long text, you should specify a pixel-based `width` property. If the text might change and you want to ensure that the Text control always takes up the same space in your application, set explicit `height` and `width` properties that fit the largest expected text.

MX TextArea control

The TextArea control is part of both the MX and Spark component sets. While you can use the MX controls in your application, Adobe recommends that you use the Spark controls instead. For more information about using Spark text controls, see "Spark text controls" on page 734.

The MX TextArea control is a multiline, editable text field with a border and optional scroll bars. The MX TextArea control supports the HTML and rich text rendering capabilities of Flash Player and AIR. The MX TextArea control dispatches `change` and `textInput` events.

The following image shows an MX TextArea control:



Congratulations.
You are a winner.

To create a single-line, editable text field, use the MX TextInput control. For more information, see "MX TextInput control" on page 825. To create noneditable text fields, use the MX Label and Text controls. For more information, see "MX Label control" on page 824 and "MX Text control" on page 826.

If you disable an MX TextArea control, it displays its contents in a different color, represented by the `disabledColor` style. You can set an MX TextArea control's `editable` property to `false` to prevent editing of the text. You can set an MX TextArea control's `displayAsPassword` property to conceal input text by displaying characters as asterisks.

Creating an MX TextArea control

You define an MX TextArea control in MXML by using the `<mx:TextArea>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/TextAreaControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:TextArea id="textConfirm"
        width="300" height="100"
        text="Please enter your thoughts here."/>
</s:Application>
```

Just as you can for the Text control, you use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see "Using the text property" on page 807 and "Using the htmlText property" on page 810.

Sizing the MX TextArea control

The MX TextArea control does not resize to fit the text that it contains. If the new text exceeds the capacity of the MX TextArea control and the `horizontalScrollPolicy` is `true` (the default value), the control adds a scrollbar.

MX RichTextEditor control

The MX RichTextEditor control lets users enter, edit, and format text. Users apply text formatting and URL links by using subcontrols that are located at the bottom of the RichTextEditor control.

There is no Spark equivalent of the RichTextEditor control.

For complete reference information, see the *Adobe Flex Language Reference*.

About the RichTextEditor control

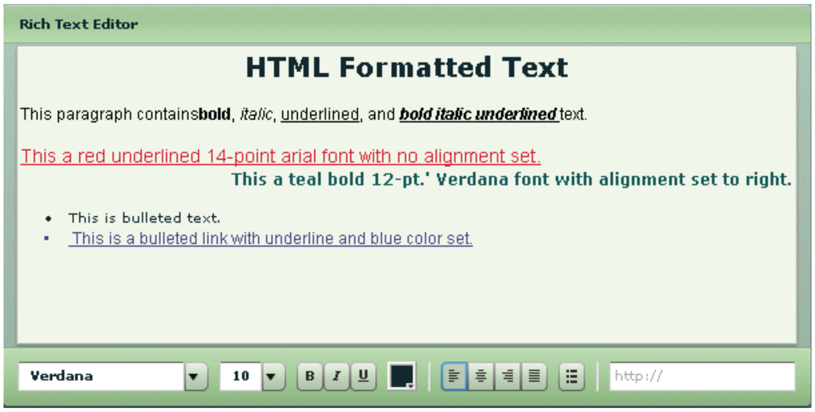
The RichTextEditor control consists of a Panel control with two direct children:

- An MX TextArea control in which users can enter text

- A tool bar container with format controls that let a user specify the text characteristics. Users can use the tool bar subcontrols to apply the following text characteristics:
  - Font family
  - Font size

    ***Note:** When using the RichTextEditor control, you specify the actual pixel value for the font size. This size is not equivalent to the relative font sizes specified in HTML by using the* `size` *attribute of the HTML* `<font>` *tag.*
  - Any combination of bold, italic and underline font styles
  - Text color
  - Text alignment: left, center, right, or justified
  - Bullets
- URL links

The following image shows a RichTextEditor control with some formatted text:



Rich Text Editor

HTML Formatted Text

This paragraph contains**bold**, *italic*, underlined, and ***bold italic underlined*** text.

This a red underlined 14-point arial font with no alignment set.

This a teal bold 12-pt.' Verdana font with alignment set to right.

- This is bulleted text.
- This is a bulleted link with underline and blue color set.

For the source for this example, see “Creating a RichTextEditor control” on page 829.

You use the RichTextEditor interactively as follows:

- Text that you type is formatted as specified by the control settings.
- To apply new formatting to existing text, select the text and set the controls to the required format.
- To create a link, select a range of text, enter the link target in the text box on the right, and press Enter. You can only specify the URL; the link always opens in a _blank target. Also, creating the link does not change the appearance of the link text; you must separately apply any color and underlining.
- You can cut, copy, and paste rich text within and between Flash HTML text fields, including the RichTextEditor control’s TextArea subcontrol, by using the normal keyboard commands. You can copy and paste plain text between the TextArea and any other text application, such as your browser or a text editor.

Creating a RichTextEditor control

You define a RichTextEditor control in MXML by using the `<mx:RichTextEditor>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/RichTextEditorControl.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:RichTextEditor id="myRTE" text="Congratulations, winner!" />
</s:Application>
```

You can use the `text` property to specify an unformatted text string, or the `htmlText` property to specify an HTML-formatted string. For more information on using these properties, see "Using the text property" on page 807, and "Using the htmlText property" on page 810. For information on selecting, replacing, and formatting text that is in the control, see "Selecting and modifying text" on page 820.

The following example shows the code used to create the image in "About the RichTextEditor control" on page 828:

```
<?xml version="1.0"?>
<!-- textcontrols/RichTextEditorControlWithFormattedText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <!-- The HTML text string used to populate the RichTextEditor control's
     TextArea subcomponent. The text is on a single line. -->
  <fx:Script><![CDATA[
     [Bindable]
     public var htmlData:String="<textformat leading='2'><p align='center'><b><font
size='20'>HTML Formatted Text</font></b></p></textformat><br><textformat leading='2'><p
align='left'><font face='_sans' size='12' color='#000000'>This paragraph contains<b>bold</b>,
<i>italic</i>, <u>underlined</u>, and <b><i><u>bold italic underlined
</u></i></b>text.</font></p></textformat><br><p><u><font face='arial' size='14'
color='#ff0000'>This a red underlined 14-point arial font with no alignment
set.</font></u></p><p align='right'><font face='verdana' size='12' color='#006666'><b>This a
teal bold 12-pt.' Verdana font with alignment set to right.</b></font></p><br><li>This is
bulleted text.</li><li><font face='arial' size='12' color='#0000ff'><u> <a
href='http://www.adobe.com'>This is a bulleted link with underline and blue color
set.</a></u></font></li>";
  ]]></fx:Script>
  <!-- The RichTextEditor control. To reference a subcomponent prefix its ID with the
RichTextEditor control ID. -->
  <mx:RichTextEditor id="rte1"
     backgroundColor="#ccffcc"
     width="500"
     headerColors="[#88bb88, #bbeebb]"
     footerColors="[#bbeebb, #88bb88]"
     title="Rich Text Editor"
     htmlText="{htmlData}"
     initialize="rte1.textArea.setStyle('backgroundColor', '0xeeffee')"
  />
</s:Application>
```

Sizing the RichTextEditor control

The control does not resize in response to the size of the text in the MX TextArea control. If the text exceeds the viewable space, by default, the MX TextArea control adds scroll bars. If you specify a value for either the `height` or `width` property but not both, the control uses the default value for the property that you do not set.

If you set a `width` value that results in a width less than 605 pixels wide, the RichTextEditor control stacks the subcontrols in rows.

Programming RichTextEditor subcontrols

Your application can control the settings of any of the RichTextEditor subcontrols, such as the MX TextArea, the ColorPicker, or any of the MX ComboBox or MX Button controls that control text formatting. To refer to a RichTextEditor subcontrol, prefix the requested control's ID with the RichTextEditor control ID. For example, to refer to the ColorPicker control in a RichTextEditor control that has the ID rte1, use rte1.colorPicker.

Inheritable styles that you apply directly to a RichTextEditor control affect the underlying Panel control and the subcontrols. Properties that you apply directly to a RichTextEditor control affect the underlying Panel control only.

For more information, see the RichTextEditor in the *Adobe Flex Language Reference*.

Setting RichTextEditor subcontrol properties and styles

The following simple code example shows how you can set and change the properties and styles of the RichTextEditor control and its subcontrols. This example uses styles that the RichTextEditor control inherits from the Panel class to set the colors of the Panel control header and the tool bar container, and sets the MX TextArea control's background color in the RichTextEditor control's creationComplete event member. When users click the buttons, their click event listeners change the TextArea control's background color and the selected color of the ColorPicker control.

```
<?xml version="1.0"?>
<!-- textcontrols/RTESubcontrol.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="420">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <!-- The RichTextEditor control. To set the a subcomponent's style or property,
  fully qualify the control ID. The footerColors style sets the ControlBar colors. -->
  <mx:RichTextEditor id="rte1"
     backgroundColor="#ccffcc"
     headerColors="[#88bb88, #bbeebb]"
     footerColors="[#bbeebb, #88bb88]"
     title="Rich Text Editor"
     creationComplete="rte1.textArea.setStyle('backgroundColor','0xeeffee')"
     text="Simple sample text"/>
  <!-- Button to set a white TextArea background. -->
  <s:Button
     label="Change appearance"
     click="rte1.textArea.setStyle('backgroundColor',
'0xffffff');rte1.colorPicker.selectedIndex=27;"/>
  <!-- Button to reset the display to its original appearance. -->
  <s:Button
     label="Reset Appearance"
     click="rte1.textArea.setStyle('backgroundColor',
'0xeeffee');rte1.colorPicker.selectedIndex=0;"/>
</s:Application>
```

Removing and adding RichTextEditor subcontrols

You can remove any of the standard RichTextEditor subcontrols, such as the alignment buttons. You can also add your own subcontrols, such as a button that pops up a find-and-replace dialog box.

Remove an existing subcontrol

1 Create a function that calls the `removeChildAt` method of the editor's tool bar Container subcontrol, specifying the control to remove.

2 Call the method in the RichTextEditor control's `initialize` event listener.

The following example removes the alignment buttons from a RichTextEditor control, and shows the default appearance of a second RichTextEditor control:

```
<?xml version="1.0"?>
<!-- textcontrols/RTERemoveAlignButtons.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="700">
    <fx:Script><![CDATA[
        public function removeAlignButtons():void {
            rt1.toolbar.removeChild(rt1.alignButtons);
        }
    ]]></fx:Script>

    <s:VGroup>
        <mx:RichTextEditor id="rt1"
            title="RichTextEditor With No Align Buttons"
            creationComplete="removeAlignButtons()"/>
        <mx:RichTextEditor id="rt2"
            title="Default RichTextEditor"/>
    </s:VGroup>

</s:Application>
```

Add a new subcontrol

1 Create an ActionScript function that defines the subcontrol. Also create any necessary methods to support the control's function.

2 Call the method in the RichTextEditor control's `initialize` event listener, as in the following tag:

```
<mx:RichTextEditor id="rt" initialize="addMyControl()"
```

The following example adds a find-and-replace dialog box to a RichTextEditor control. It consists of two files: the application, and a custom TitleWindow control that defines the find-and-replace dialog (which also performs the find-and-replace operation on the text). The application includes a function that adds a button to pop up the TitleWindow, as follows:

```
<?xml version="1.0"?>
<!-- textcontrols/CustomRTE.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.controls.*;
            import mx.containers.*;
            import flash.events.*;
            import mx.managers.PopUpManager;
            import mx.core.IFlexDisplayObject;
            /* The variable for the pop-up dialog box. */
            public var w:IFlexDisplayObject;
            /* Add the Find/Replace button to the Rich Text Editor control's
               tool bar container. */
            public function addFindReplaceButton():void {
                var but:Button = new mx.controls.Button();
                but.label = "Find/Replace";
                but.addEventListener("click",findReplaceDialog);
                rt.toolbar.addChild(but);
            }
            /* The event listener for the Find/Replace button's click event
               creates a pop-up with a MyTitleWindow custom control. */
            public function findReplaceDialog(event:Event):void {
            var w:MyTitleWindow = MyTitleWindow(PopUpManager.createPopUp(this, MyTitleWindow,
true));
                w.height=200;
                w.width=340;
                /* Pass the a reference to the textArea subcomponent
                   so that the custom control can replace the text. */
                w.RTETextArea = rt.textArea;
                PopUpManager.centerPopUp(w);
            }
        ]]>
    </fx:Script>
    <mx:RichTextEditor id="rt" width="95%"
        title="RichTextEditor"
        text="This is a short sentence."
        initialize="addFindReplaceButton()"/>
</s:Application>
```

The following MyTitleWindow.mxml file defines the custom myTitleWindow control that contains the find-and-replace interface and logic:

```
<?xml version="1.0"?>
<!-- textcontrols/MyTitleWindow.mxml -->
<!-- A TitleWindow that displays the X close button. Clicking the close button
only generates a CloseEvent event, so it must handle the event to close the control. -->
<mx:TitleWindow
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    title="Find/Replace"
    showCloseButton="true"
    close="closeDialog();">

    <fx:Script>
        <![CDATA[
            import mx.controls.TextArea;
            import mx.managers.PopUpManager;
            /* Reference to the RichTextArea textArea subcomponent.
               It is set by the application findReplaceDialog method
               and used in the replaceAndClose method, below. */
            public var RTETextArea:TextArea;
            /* The event handler for the Replace button's click event.
               Replace the text in the RichTextEditor TextArea and
               close the dialog box. */
            public function replaceAndClose():void{
                RTETextArea.text = RTETextArea.text.replace(ti1.text, ti2.text);
                PopUpManager.removePopUp(this);
            }
            /* The event handler for the TitleWindow close button. */
            public function closeDialog():void {
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
    <!-- The TitleWindow subcomponents: the find and replace inputs,
         their labels, and a button to initiate the operation. -->
    <mx:Label text="Find what:"/>
    <mx:TextInput id="ti1"/>

    <mx:Label text="Replace with:"/>
    <mx:TextInput id="ti2"/>

    <mx:Button label="Replace" click="replaceAndClose();"/>
</mx:TitleWindow>
```

Using the Flex AIR components

When building an Adobe® AIR® application in Flex, you can use any of the controls and other components that are part of Flex. In addition, Flex includes a set of components that are specifically for AIR applications.

Flex AIR components overview

The Flex AIR components can be divided into the following groups:

File system controls

The file system controls are a set of user-interface controls that provide information about and tools to interact with the file system of the local computer on which the AIR application is running. These include controls for displaying lists of files in tree or grid format, controls for choosing directories or files from a list or combo box, and so on.

- "About file system controls" on page 835
- "FileSystemComboBox control" on page 836
- "FileSystemTree control" on page 837
- "FileSystemList control" on page 838
- "FileSystemDataGrid control" on page 839
- "FileSystemHistoryButton control" on page 840
- "Displaying a directory structure with Flex AIR" on page 841

HTML control

The HTML control is used to display an HTML web page within a Flex application. For example, you could use it to combine HTML and Flex content in a single application. You cannot use the HTML control in a mobile Flex application.

- "About the HTML control" on page 842

FlexNativeMenu control

The FlexNativeMenu control provides the ability to use MXML to declaratively define the structure of a native menu. You can use it to define an application menu (on OS X), a window's menu (on Windows), a context menu, and so forth.

- "About the FlexNativeMenu control" on page 843

Window containers

The window containers are two components that can be used as containers for defining the layout of windows in applications. There are two window containers: the WindowedApplication, a substitute for the Application container to use as the main or initial window of an AIR application; and the Window, for application windows that are opened after the initial window of the application.

- "About the AIR window containers" on page 861.
- "WindowedApplication container" on page 861
- "Window container" on page 862

About file system controls

The Flex file system components combine the functionality of other Flex controls, such as Tree, DataGrid, ComboBox, and so forth, with pre-built awareness of the file system on the application user's computer. These controls duplicate the functionality of user interface controls that are commonly used in desktop applications for browsing and selecting files and directories. You can use one or two of them to directly include file-related functionality in a screen of your application. Or you can combine several of them together to create a full-featured file browsing or selection dialog box.

Each of the Flex file system controls, except the FileSystemHistoryButton control, displays a view of the contents of a particular directory in the computer's file system. For instance, the FileSystemTree displays the directory's contents in a hierarchical tree (using the Flex Tree control) and the FileSystemComboBox displays the directory and its parent directories in the menu of a ComboBox control.

***Note:** The File class includes methods for accessing the file and directory selection dialog boxes built-in to the operating system. For example, the File.browseForOpen() method presents the user with a dialog box for opening a file. The File class also includes the `browseForDirectory()`, `browseForOpenMultiple()`, and `browseForSave()` methods. It is often better and easier to use these methods that to construct your own file selection user interface. For more information, see Letting the user browse to select a file and Letting the user browse to select a directory.*

For any of the Flex file system controls except the FileSystemHistoryButton control, you use the control's `directory` property to change the currently selected directory for a control. You can also use the `directory` property to retrieve the current directory, such as if the user selects a directory in the control.

FileSystemComboBox control

A FileSystemComboBox defines a combo box control for selecting a location in a file system. The control always displays the selected directory in the combo box's text field. When the combo box's drop-down list is displayed, it shows the hierarchy of directories that contain the selected directory, up to the computer root directory. The user can select a higher-level directory from the list. In this sense, the FileSystemComboBox control's behavior is different from the FileSystemTree, FileSystemList, and FileSystemDataGrid controls that display the directories and files that are contained by the current directory.

For more information on the FileSystemComboBox control, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FileSystemComboBox control

You use the `<mx:FileSystemComboBox>` tag to define a FileSystemComboBox control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. The `directory` property can be set in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property, the data provider of the underlying combo box is automatically populated. By default the `directory` property is set to the root "Computer" directory, which has no ancestor directories and hence shows no selectable directories in the combo box's drop-down list.

The following example shows four variations on the basic FileSystemComboBox. Each combo box is initially set to the user's desktop directory, in the application's `creationComplete` handler. The distinct characteristics of the combo boxes are as follows:

- The first combo box simply displays the selected directory.
- The second combo box's `showIcons` property is set to `false`, so no icon appears next to the items in the combo box's list.
- The third combo box's `indent` property is set to 20, which is larger than the default. As a result, the items in the combo box's list are more indented than normal.
- The fourth combo box has an event handler defined for the `directoryChange` event. When the selected directory in the combo box changes, it calls the `setOutput()` method, which writes the selected directory's path to a TextArea control named `output`.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/FSComboBoxSimple.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.filesystem.File;

            private function init():void {
                fcb.directory = File.desktopDirectory;
                fcbIndent.directory = File.desktopDirectory;
                fcbNoIcons.directory = File.desktopDirectory;
                fcbChange.directory = File.desktopDirectory;
            }

            private function setOutput():void {
                output.text = fcbChange.directory.nativePath;
            }
        ]]>
    </fx:Script>

    <mx:FileSystemComboBox id="fcb"/>
    <mx:FileSystemComboBox id="fcbNoIcons"
        showIcons="false"/>
    <mx:FileSystemComboBox id="fcbIndent"
        indent="20"/>
    <mx:FileSystemComboBox id="fcbChange"
        directoryChange="setOutput();"/>
    <s:TextArea id="output"
        width="200" height="50"/>
</s:WindowedApplication>
```

FileSystemComboBox user interaction

The FileSystemComboBox supports the same user interaction as a standard combo box control. The control displays a directory in its selection field. The user clicks the button (or uses the keyboard) to open a drop-down list containing the names of the hierarchy of directories that contain the selected directory. The user can then select one of the directories, which causes the drop-down list to close and the selected directory to become the current directory. When the user selects a directory, the control dispatches the `directoryChange` event, and its `directory` property changes to the newly selected directory.

FileSystemTree control

A FileSystemTree control displays the contents of a file system directory as a tree. The tree can display the directory's files, its subdirectories, or both. For files, file names can be displayed with or without extensions.

For more information on the FileSystemTree control, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FileSystemTree control

You use the `<mx:FileSystemTree>` tag to define a FileSystemTree control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. You can set the directory property in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property, the data provider of the underlying tree control is automatically populated. The specified directory isn't displayed in the tree—its child files or directories are shown as the top-level nodes of the tree. By default the `directory` property is set to the root "Computer" directory. Consequently, its children (the drive or drives attached to the computer) are displayed as the top branches of the tree.

The following example demonstrates creating a FileSystemTree control that displays all files and folders, which is the default. In addition, hidden files are shown by setting the `showHidden` property to `true`.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/FSTreeSimple.mxml -->
<s:WindowedApplication
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:FileSystemTree showHidden="true"/>
</s:WindowedApplication>
```

FileSystemTree user interaction

The FileSystemTree control supports the same types of user interaction as the standard Flex Tree control. In addition, the FileSystemTree control provides several additional events for file-related functionality. If the user double-clicks a closed directory node or clicks its disclosure icon, the control dispatches a `directoryOpening` event. If the user double-clicks an open directory node or clicks its disclosure icon, the control dispatches a `directoryClosing` event. If the user double-clicks a file node, the control dispatches a `fileChoose` event.

FileSystemList control

A FileSystemList control displays the contents of a file system directory as selectable items in a scrolling list (a Flex List control). The displayed contents can include subdirectories and files, with additional filtering options as well. A FileSystemList control can be linked to a FileSystemHistoryButton control, meaning that the button can be used to move to a previously displayed directory.

For more information on the FileSystemList control, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FileSystemList control

You use the `<mx:FileSystemList>` tag to define a FileSystemList control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. The directory property can be set in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property, the data provider of the underlying list control is automatically populated. The specified directory isn't displayed in the list—its child files or directories are shown as the items in the list. By default the `directory` property is set to the root "Computer" directory. In that case its children, which are the drive or drives attached to the computer, are displayed as the items in the list.

The following example demonstrates creating a FileSystemList control that displays all files and folders (the default). The sample also includes a button for navigating up one level in the directory hierarchy. The button is enabled if the currently displayed directory has a parent directory because the button's `enabled` property is bound to the FileSystemList control's `canNavigateUp` property. The button navigates up one level by calling the FileSystemList control's `navigateUp()` method when it is clicked.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/FSListSimple.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:FileSystemList id="fileList"/>
    <s:Button label="Up"
        click="fileList.navigateUp();"
        enabled="{fileList.canNavigateUp}"/>
</s:WindowedApplication>
```

FileSystemList user interaction

The FileSystemList control provides standard scrolling list functionality for files: a user can scroll through the list of files and select one or multiple files or directories. When the user double-clicks a directory, the FileSystemList control automatically sets that directory as the control's `directory` property. It then becomes the directory whose contents are displayed in the list.

FileSystemDataGrid control

A FileSystemDataGrid displays file information in a data-grid format. The file information displayed includes the file name, creation date, modification date, type, and size. Data grid columns displaying this information are automatically created in the underlying DataGrid control, and can be removed or customized in the same way that you customize DataGrid columns. The displayed contents can include subdirectories and files, with additional filtering options as well. A FileSystemList control can be linked to a FileSystemHistoryButton control, meaning that the button can be used to move to a previously displayed directory.

For more information on the FileSystemDataGrid control, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FileSystemDataGrid control

You use the `<mx:FileSystemDataGrid>` tag to define a FileSystemDataGrid control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. You can set the directory property in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property the data provider of the underlying data grid control is automatically populated. The specified directory isn't displayed in the grid—its child files or directories are shown as the rows in the grid. By default the `directory` property is set to the root "Computer" directory. In that case its children, the drive or drives attached to the computer, are displayed as the items in the grid.

The following example demonstrates creating a FileSystemDataGrid control that displays all files and folders (the default). The sample also includes a button for navigating up one level in the directory hierarchy. The button is enabled if the currently displayed directory has a parent directory because the button's `enabled` property is bound to the FileSystemDataGrid control's `canNavigateUp` property. The button navigates up one level by calling the FileSystemDataGrid control's `navigateUp()` method when it is clicked.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/FSDataGridSimple.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <mx:FileSystemDataGrid id="fileGrid"/>
    <s:Button label="Up"
        click="fileGrid.navigateUp();"
        enabled="{fileGrid.canNavigateUp}"/>
</s:WindowedApplication>
```

FileSystemDataGrid user interaction

The FileSystemDataGrid control includes standard DataGrid functionality such as scrolling through the grid, selecting grid rows, reordering grid columns, and sorting grid data by clicking the grid headers.

In addition, the FileSystemDataGrid provides some file-specific functionality. A FileSystemDataGrid allows a user to navigate to other directories using the mouse or keyboard. The user can change the directory by double-clicking a subdirectory, by pressing Enter or Ctrl-Down when a subdirectory is selected, by pressing Ctrl-Up when the control isn't displaying the COMPUTER directory, by pressing Ctrl-Left when there is a “previous” directory to navigate back to, or by pressing Ctrl-Right when there is a “next” directory to navigate forward to.

If the user attempts to change the directory being displayed, the control dispatches a cancelable `directoryChanging` event. If the event isn’t canceled, the control displays the contents of the new directory and its `directory` property changes. Whenever the `directory` property changes for any reason, the controls dispatches a `directoryChange` event.

FileSystemHistoryButton control

The FileSystemHistoryButton control lets the user move backwards or forwards through the navigation history of another control. It works in conjunction with a FileSystemList or FileSystemDataGrid control, or any similar control with a property containing an array of File objects. The FileSystemHistoryButton is a PopUpMenuButton. It has a button for navigating back or forward one step in the history. It also has a list of history steps from which one step can be chosen.

To link a FileSystemHistoryButton to a control, bind the button’s `dataProvider` property to one of the control’s properties. The property must contain an array of File objects representing a sequence of directories in a file system browsing history. For instance, you can bind the `dataProvider` property to the `forwardHistory` or `backHistory` property of a FileSystemList or FileSystemDataGrid control. The button can then be used to navigate the display history of that control if you set the `click` and `itemClick` event handlers of the button to call the `navigateForward()` or `navigateBack()` method of the control.

For more information on the FileSystemHistoryButton control, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FileSystemHistoryButton control

You use the `<mx:FileSystemHistoryButton>` tag to define a FileSystemHistoryButton control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the property to which the button is bound by setting the `dataProvider` property.

The following example demonstrates creating two FileSystemHistoryButton controls that are linked to the display history of a FileSystemList control. Each button's `enabled` property is bound to the FileSystemList control's `canNavigateBack` or `canNavigateForward` property. As a result, the button is enabled if the currently displayed directory can navigate in the appropriate direction. When the user clicks a button, its event listener calls the FileSystemList control's `navigateBack()` or `navigateForward()` method. This causes the FileSystemList control to navigate to the previous or next directory.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/FSHistoryBSimple.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:HGroup>
        <mx:FileSystemHistoryButton label="Back"
            dataProvider="{fileList.backHistory}"
            enabled="{fileList.canNavigateBack}"
            click="fileList.navigateBack();"
            itemClick="fileList.navigateBack(event.index)"/>
        <mx:FileSystemHistoryButton label="Forward"
            dataProvider="{fileList.forwardHistory}"
            enabled="{fileList.canNavigateForward}"
            click="fileList.navigateForward();"
            itemClick="fileList.navigateForward(event.index)"/>
    </s:HGroup>
    <mx:FileSystemList id="fileList"/>
</s:WindowedApplication>
```

FileSystemHistoryButton user interaction

The FileSystemHistoryButton is based on the Flex PopUpMenuButton, so their core functionality is the same. When the user clicks the main button the click event is dispatched (normally moving backward or forward one step in the history). In addition, by clicking the pull-down menu button, a list of the history steps is displayed. This allows the user to navigate directly to a specific step in the history.

Displaying a directory structure with Flex AIR

The following example uses the WindowedApplication container and the FileSystemTree and FileSystemDataGrid controls. In this example, the FileSystemTree control displays a directory structure. Clicking a directory name in the FileSystemTree control causes the FileSystemDataGrid control to display information about the files in the selected directory:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/FSDirApp.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="750" height="500">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <mx:HDividedBox>
        <mx:FileSystemTree id="tree"
            width="200" height="100%"
            directory="{new File('C:\\')}"
            enumerationMode="directoriesOnly"
            change="dataGrid.directory = File(tree.selectedItem);"/>
        <mx:FileSystemDataGrid id="dataGrid"
            width="100%" height="100%"
            directory="{new File('C:\\')}"/>
    </mx:HDividedBox>
</s:WindowedApplication>
```

About the HTML control

An HTML control displays HTML web pages in your application. It is designed to be used to render specific external HTML content within your AIR application. It offers functionality like a lightweight web browser, including loading HTML pages, navigation history, and the ability to access the raw HTML content.

You cannot use the HTML control in a mobile Flex application. To show HTML in a mobile Flex application, use the StageWebView class instead. Blogger Judah Frangipane describes how to do this. There is also an Adobe Developer Connection article that shows you how to create a basic web browser in a mobile Flex application.

The HTML control is not designed or intended to be used as a replacement for the Text or TextArea controls. Those controls are more appropriate for displaying formatted text or for use as an item renderer for displaying short runs of text.

Creating an HTML control

You use the `<mx:HTML>` tag to define an HTML control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the location of the HTML page to display by setting the `location` property.

The following example demonstrates the use of an HTML control in a simple application. The HTML control's `location` property is set to "http://labs.adobe.com/", so that URL is opened in the control when it loads. In addition, when the "back" and "forward" are clicked they call the control's `historyBack()` and `historyForward()` methods. A TextInput control allows the user to enter a URL location. When a third "go" button is clicked, the HTML control's `location` property is set to the `text` property of the input text field.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/HTMLSimple.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:ControlBar width="100%">
        <s:Button label="&lt; Back"
            click="content.historyBack();"/>
        <s:Button label="Forward &gt;"
            click="content.historyForward();"/>
        <s:TextInput id="address"
            text="{content.location}" width="100%"/>
        <s:Button label="Go!"
            click="content.location = address.text"/>
    </mx:ControlBar>
    <s:Group width="100%" height="100%">
        <mx:HTML id="content" location="http://labs.adobe.com/"/>
    </s:Group>
</s:WindowedApplication>
```

HTML control user interaction

For a user interacting with an HTML control, the experience is like using a web browser with only the content window and no menu bar or navigation buttons. The HTML page content displays in the control. The user can interact with the content through form fields and buttons and by clicking hyperlinks. Some of these interactions, such as clicking a link or submitting a form, would normally cause a browser to load a new page. These actions cause the HTML control to display the content of the new page and also change the value of the control's `location` property.

About the FlexNativeMenu control

A FlexNativeMenu component is a Flex wrapper for the NativeMenu class. The FlexNativeMenu allows you to use MXML and a data provider to define the structure of a menu. The FlexNativeMenu component does not have any visual representation that is rendered by Flex. Instead, a FlexNativeMenu instance defines a native operating system menu such as an application menu (OS X), a window menu (Windows), a context menu, or any other native menu that can be created in AIR.

The FlexNativeMenu component is designed to be like the Flex Menu and MenuBar components. Developers who have worked with those components should find the FlexNativeMenu familiar.

For more information on the FlexNativeMenu control, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a FlexNativeMenu control

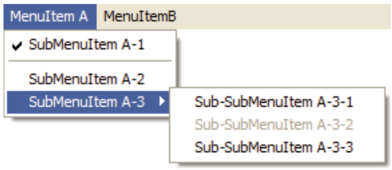
You define a FlexNativeMenu control in MXML by using the `<mx:FlexNativeMenu>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block. Because the FlexNativeMenu control have any visual representation, if you want to use it as the direct child of a container in MXML, you must insert it in an `<fx:Declaration>` block.

You specify the data for the FlexNativeMenu control by using the `dataProvider` property. The FlexNativeMenu control uses the same types of data providers as does the MenuBar control and the Menu control. Several of the XML attributes or object property names have meaning to the FlexNativeMenu control. For more information on structuring FlexNativeMenu data providers, see "Defining FlexNativeMenu menu structure and data" on page 849.

You can assign any name to node tags in the XML data. In subsequent examples, each node is named with the generic `<menuitem>` tag, but you can use `<node>`, `<subNode>`, `<person>`, `<address>`, and so on.

Creating an application or window menu

When you create an application or window menu using the FlexNativeMenu control, the top-level objects or nodes in the data provider correspond to the top-level menu items. In other words, they define the items that display in the menu bar itself. Items nested inside one of those top-level items define the items within the menu. Likewise, those menu items can contain items, in which case the menu item is a submenu. When the user selects the menu item it expands its own menu items. For example, the following image shows a window menu with three menu items (plus an additional separator menu item). The item with the label "SubMenuItem A-3" in turn contains three menu items, so SubMenuItem A-3 is treated as a submenu. (The code to create this menu is provided later.)



For an MXML application using the Flex WindowedApplication container as the root MXML node, you can assign a FlexNativeMenu to the WindowedApplication instance's `menu` property. The menu is used as the application menu on OS X and the window menu of the initial window on Windows. Likewise, to specify a window menu for an additional window defined using the Flex Window container, assign a FlexNativeMenu to the Window instance's `menu` property. In that case the menu displays on Windows only and is ignored on OS X.

***Note:** Mac OS X defines a menu containing standard items for every application. Assigning a FlexNativeMenu object to the* `menu` *property of the WindowedApplication component replaces the standard menu rather than adding additional menus to it.*

The following application defines a FlexNativeMenu as the `menu` property of a WindowedApplication container. Consequently, the specified menu is used as the application menu on Mac OS and the window menu of the initial window on Windows. This code creates the menu shown in the previous image:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMSimple.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:menu>
        <mx:FlexNativeMenu dataProvider="{myMenuData}"
            labelField="@label"
            showRoot="false"/>
    </s:menu>

    <fx:Declarations>
        <fx:XML format="e4x" id="myMenuData">
            <root>
                <menuitem label="MenuItem A">
                    <menuitem label="SubMenuItem A-1" type="check" toggled="true"/>
                    <menuitem type="separator"/>
                    <menuitem label="SubMenuItem A-2"/>
                    <menuitem label="SubMenuItem A-3">
                        <menuitem label="Sub-SubMenuItem A-3-1"/>
                        <menuitem label="Sub-SubMenuItem A-3-2" enabled="false"/>
                        <menuitem label="Sub-SubMenuItem A-3-3"/>
                    </menuitem>
                </menuitem>
                <menuitem label="MenuItemB">
                    <menuitem label="SubMenuItem B-1"/>
                    <menuitem label="SubMenuItem B-2"/>
                </menuitem>
            </root>
        </fx:XML>
    </fx:Declarations>
</s:WindowedApplication>
```

in this example, the FlexNativeMenu control is used as the value of the `WindowedApplication.menu` property, so it does not have to be in an `<fx:Declaration>` block. However, if it was the direct child of the WindowedApplication container, it must be in an `<fx:Declaration>` block. For an example defining the FlexNativeMenu control in an `<fx:Declaration>` block, see “Handling FlexNativeMenu control events” on page 858.

Creating a context menu

Creating a context menu in a Flex AIR application involves two steps. You create the FlexNativeMenu instance that defines the menu structure. You then assign that menu as the context menu for its associated control. Because a context menu consists of a single menu, the top-level menu items serve as the items in the single menu. Any menu item that contains child menu items defines a submenu within the single context menu.

The FlexNativeMenu is a replacement for the context menu that you use with browser-based Flex applications (the flash.ui.ContextMenu class). You can use one type of menu or the other, but you can’t specify both types for a single component.

To assign a FlexNativeMenu component as the context menu for a visual Flex control, call the FlexNativeMenu instance’s `setContextMenu()` method, passing the visual control as the `component` parameter (the only parameter):

```
menu.setContextMenu(someComponent);
```

The same FlexNativeMenu can be used as the context menu for more than one object, by calling `setContextMenu()` multiple times using different `component` parameter values. You can also reverse the process (that is, remove an assigned context menu) using the `unsetContextMenu()` method.

The following example demonstrates creating a FlexNativeMenu component and setting it as the context menu for a Label control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMList.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">

    <fx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void
            {
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;
                myMenu.setContextMenu(lbl);
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <!-- The XML data provider -->
        <fx:XML format="e4x" id="menuData">
            <root>
                <menuitem label="MenuItem A"/>
                <menuitem label="MenuItem B"/>
                <menuitem label="MenuItem C"/>
            </root>
        </fx:XML>
    </fx:Declarations>

    <mx:Label id="lbl" x="100" y="10"
        text="Right-click here to open menu"/>
</s:WindowedApplication>
```

In addition to context menus for visual components within an application window, an AIR application supports two other special context menus: dock icon menus (OS X) and system tray icon menus (Windows). To set either of these menus, you define the menu's structure using the FlexNativeMenu component, then you assign the FlexNativeMenu instance to the WindowedApplication container's `dockIconMenu` or `systemTrayIconMenu` property.

Before setting the `dockIconMenu` or `systemTrayIconMenu` property you may want to determine whether the user's operating system supports a dock icon or a system tray icon, using the NativeApplication class's static `supportsDockIcon` and `supportsSystemTrayIcon` properties. Doing so isn't necessary, but can be useful. For instance, you might want to customize a menu depending on whether it is used as the context menu for a dock icon or for a system tray icon.

Finally, while a dock icon exists automatically for an application, you must explicitly specify a system tray icon in order for the icon to appear. (Naturally, the icon must exist in order for the user to be able to right-click the icon to activate the context menu).

The following example defines a FlexNativeMenu that is used as a context menu. If the user's operating system supports a system tray icon, the code creates an icon and displays it in the system tray. The code then assigns the FlexNativeMenu instance as the context menu for the system tray and dock icon menus.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMContextM.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import flash.desktop.DockIcon;
            import flash.desktop.InteractiveIcon;
            import flash.desktop.NativeApplication;
            import flash.desktop.SystemTrayIcon;
            import flash.display.Shape;
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void {
                // Create the menu
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;

                var icon:InteractiveIcon;
                icon = NativeApplication.nativeApplication.icon;

                // If we need a system tray icon, create one and display it
                if (NativeApplication.supportsSystemTrayIcon)
                {
                    var iconData:BitmapData = createSystemTrayIcon();
                    SystemTrayIcon(icon).bitmaps = new Array(iconData);
                }

                // Use this approach if you want to assign the same menu
                // to the dock icon and system tray icon
                this.systemTrayIconMenu = this.dockIconMenu = myMenu;

                // Use this approach if you want to assign separate menus
//              if (NativeApplication.supportsDockIcon)
//              {
//                  this.dockIconMenu = myMenu;
//              }
//              else if (NativeApplication.supportsSystemTrayIcon)
//              {
//                  this.systemTrayIconMenu = myMenu;
//              }
            }
            private function createSystemTrayIcon():BitmapData {
                // Draw the icon in a Graphic
                var canvas:Shape = new Shape();
                canvas.graphics.beginFill(0xffff00);
                canvas.graphics.drawCircle(24, 24, 24);
                canvas.graphics.endFill();
                canvas.graphics.beginFill(0x000000);
                canvas.graphics.drawEllipse(13, 13, 9, 12);
                canvas.graphics.drawEllipse(27, 13, 9, 12);
                canvas.graphics.endFill();
                canvas.graphics.lineStyle(3, 0x000000);
                canvas.graphics.moveTo(11, 32);
                canvas.graphics.curveTo(24, 46, 37, 32);

                var result:BitmapData = new BitmapData(48, 48, true, 0x00000000);
                result.draw(canvas);

                return result;
            }
        ]]>
    </fx:Script>

     <fx:Declarations>
        <!-- The XML data provider -->
        <fx:XML format="e4x" id="menuData">
            <root>
                <menuitem label="MenuItem A"/>
                <menuitem label="MenuItem B"/>
                <menuitem label="MenuItem C"/>
            </root>
        </fx:XML>
     </fx:Declarations>

    <s:Label text="Right-click on the dock icon (Mac OS X) or system tray icon (Windows)"/>
</s:WindowedApplication>
```

***Note:** Mac OS X defines a standard menu for the application dock icon. When you assign a FlexNativeMenu as the dock icon's menu, the items in that menu are displayed above the standard items. You cannot remove, access, or modify the standard menu items.*

Creating a pop-up menu

A pop-up menu is like a context menu, but the pop-up menu isn't necessarily associated with a particular Flex component. To open a pop-up menu, create a FlexNativeMenu instance and set its `dataProvider` property to populate the menu. To open the menu on the screen, call its `display()` method:

```
myMenu.display(this.stage, 10, 10);
```

The `display()` method has three required parameters: the Stage instance that defines the coordinates within which the menu is placed, the x coordinate where the menu is placed, and the y coordinate for the menu. For an example of using the `display()` method to create a pop-up menu, see "Example: An Array FlexNativeMenu data provider" on page 852.

One important thing to keep in mind is that the `display()` method operates immediately when it's called. Several property changes cause the FlexNativeMenu's data provider to invalidate (such as changes to the data provider, changing the `labelField`, and so forth). When the `display()` method is called immediately after making such changes, those changes aren't reflected in the menu that appears on the screen. For example, in the following code listing when the button is clicked no menu appears because the menu is created and the data provider is specified in the same block of code in which the `display()` method is called:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMDisplay.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private function createAndShow():void
            {
                var myMenu:FlexNativeMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;
                // calling display() here has no result, because the data provider
                // has been set but the underlying NativeMenu hasn't been created yet.
                myMenu.display(this.stage, 10, 10);
            }
        ]]>
    </fx:Script>

     <fx:Declarations>
        <!-- The XML data provider -->
        <fx:XML format="e4x" id="menuData">
            <root>
                <menuitem label="MenuItem A"/>
                <menuitem label="MenuItem B"/>
                <menuitem label="MenuItem C"/>
            </root>
        </fx:XML>
     </fx:Declarations>
    <!-- Button control to create and open the menu. -->
    <s:Button x="300" y="10"
        label="Open Menu"
        click="createAndShow();"/>
</s:WindowedApplication>
```

Defining FlexNativeMenu menu structure and data

The techniques for defining structure and data for a FlexNativeMenu are like the techniques for structuring all Flex menu controls. Consequently, this section does not provide comprehensive information on structuring Flex menus, but instead focuses on differences between FlexNativeMenu structure versus other Flex menu components.

The `dataProvider` property of a FlexNativeMenu defines the structure of the menu. To change a menu's structure at runtime, change the data provider and the menu updates itself accordingly. Menus typically use a hierarchical data provider such as nested arrays or XML. However, a simple menu may consist of a single flat structure of menu items.

A FlexNativeMenu instance uses a data descriptor to parse and manipulate the data provider's contents. By default, a FlexNativeMenu control uses a DefaultDataDescriptor instance as its descriptor. However, you can customize menu data parsing by creating your own data descriptor class and setting it as the FlexNativeMenu control's `dataDescriptor` property. The DefaultDataDescriptor supports a data provider that is an XML object or XMLList object, an array of objects, an object with a `children` property containing an array of objects, or a collection that implements the ICollectionView interface such as an ArrayCollection or XMLListCollection instance.

Specifying and using menu entry information

Information in a FlexNativeMenu control's data provider determines how each menu entry appears and is used. To access or change the menu contents, you modify the contents of the data provider.

The FlexNativeMenu class uses the methods of the IMenuDataDescriptor interface to access and manipulate information in the data provider that defines the menu behavior and contents. Flex includes the DefaultDataDescriptor class that implements this interface. A FlexNativeMenu control uses the DefaultDataDescriptor class if you do not specify another class in the `dataDescriptor` property.

Menu entry types

Each data provider entry can specify an item type and type-specific information about the menu item. Menu-based classes support the following item types (`type` field values):

Menu item type	Description
normal	The default type. Selecting an item with the `normal` type triggers an `itemClick` event. Alternatively, if the item has children, the menu dispatches a `menuShow` event and opens a submenu.
check	Selecting an item with the `check` type toggles the menu item's `toggled` property between `true` and `false` values and triggers an `itemClick` event. When the menu item is in the `true` state, it displays a check mark in the menu next to the item's label.
separator	Items with the `separator` type provide a simple horizontal line that divides the items in a menu into different visual groups.

Unlike other Flex menu controls, the FlexNativeMenu component does not support radio-button menu items (type `radio`).

Menu attributes

Menu items can specify several attributes that determine how the item is displayed and behaves. The following table lists the attributes you can specify, their data types, their purposes, and how the data provider must represent them if the menu uses the DefaultDataDescriptor class to parse the data provider:

Attribute	Type	Description
`altKey`	Boolean	Specifies whether the Alt key is required as part of the key equivalent for the item.
`commandKey`	Boolean	Specifies whether the Command key is required as part of the key equivalent for the item.
`controlKey`	Boolean	Specifies whether the Control key is required as part of the key equivalent for the item.
`enabled`	Boolean	Specifies whether the user can select the menu item (`true`), or not (`false`). If not specified, Flex treats the item as if the value were `true`. If you use the default data descriptor, data providers must use an `enabled` XML attribute or object field to specify this characteristic.

Attribute	Type	Description
keyEquivalent	String	Specifies a keyboard character which, when pressed, triggers an event as though the menu item was selected. The menu's keyEquivalentField or keyEquivalentFunction property determines the name of the field in the data that specifies the key equivalent, or a function for determining the key equivalents. (If the data provider is in E4X XML format, you must specify one of these properties to assign a key equivalent.)
label	String	Specifies the text that appears in the control. This item is used for all menu item types except separator. The menu's labelField or labelFunction property determines the name of the field in the data that specifies the label, or a function for determining the labels. (If the data provider is in E4X XML format, you must specify one of these properties to display a label.) If the data provider is an array of strings, Flex uses the string value as the label.
mnemonicIndex	Integer	Specifies the index position of the character in the label that is used as the mnemonic for the menu item. The menu's mnemonicIndexField or mnemonicIndexFunction property determines the name of the field in the data that specifies the mnemonic index, or a function for determining mnemonic index. (If the data provider is in E4X XML format, you must specify one of these properties to specify a mnemonic index in the data.) Alternatively, you can indicate that a character in the label is the menu item's mnemonic by including an underscore immediately to the left of that character.
shiftKey	String	Specifies whether the Shift key is required as part of the key equivalent for the item.
toggled	Boolean	Specifies whether a check item is selected. If not specified, Flex treats the item as if the value were false and the item is not selected. If you use the default data descriptor, data providers must use a toggled XML attribute or object field to specify this characteristic.
type	String	Specifies the type of menu item. Meaningful values are separator and check. Flex treats all other values, or nodes with no type entry, as normal menu entries. If you use the default data descriptor, data providers must use a type XML attribute or object field to specify this characteristic.

Unlike other Flex menu controls, the FlexNativeMenu component does not support the groupName or icon attributes. In addition, it supports the additional attribute keyEquivalent and the key equivalent modifier attributes altKey, commandKey, controlKey, and shiftKey.

The FlexNativeMenu component ignores all other object fields or XML attributes, so you can use them for application-specific data.

Considerations for XML-based FlexNativeMenu data providers

In a simple case for creating a single menu or menu bar using the FlexNativeMenu control, you might use an <fx:XML> or <fx:XMLList> tag and standard XML node syntax to define the menu data provider. When you use an XML-based data provider, keep the following rules in mind:

- With the <fx:XML> tag you must have a single root node, and you set the showRoot property of the FlexNativeMenu control to false. (Otherwise, your FlexNativeMenu would have only the root node as a menu item.) With the <fx:XMLList> tag you define a list of XML nodes, and the top-level nodes define the top-level menu items.
- If your data provider has label, keyEquivalent, or mnemonicIndex attributes, the default attribute names are not recognized by the DefaultDataDescriptor class. Set the FlexNativeMenu control's labelField , keyEquivalentField, or mnemonicIndexField property and use the E4X @ notation to specify the attribute name, such as:

```
  labelField="@label"
  keyEquivalentField="@keyEquivalent"
  mnemonicIndexField="@mnemonicIndex"
```

Example: An Array FlexNativeMenu data provider

The following example uses a FlexNativeMenu component to display a popup menu. It demonstrates how to define the menu structure using an Array of plain objects as a data provider. For an application that specifies an identical menu structure in XML, see “Example: An XML FlexNativeMenu data provider” on page 853.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMArray.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">

    <fx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void {
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.showRoot = false;
            }

            // Method to show the menu.
            private function show():void {
                myMenu.display(this.stage, 10, 10);
            }

            // The Array data provider
            [Bindable]
            public var menuData:Array = [
                {label: "MenuItem A"},
                {label: "MenuItem B", type: "check", toggled: true},
                {label: "MenuItem C", enabled: false},
                {type: "separator"},
                {label: "MenuItem D", children: [
                    {label: "SubMenuItem D-1"},
                    {label: "SubMenuItem D-2"},
                    {label: "SubMenuItem D-3"}
                    ]}
                ];
        ]]>
    </fx:Script>

    <!-- Button control to create and open the menu. -->
    <s:Button x="300" y="10"
        label="Open Menu"
        click="show();"/>
</s:WindowedApplication>
```

Example: An XML FlexNativeMenu data provider

The following example displays a popup menu using a FlexNativeMenu component. It shows how to define the menu structure using XML as a data provider. For an application that specifies an identical menu structure using an Array of objects as a data provider, see “Example: An Array FlexNativeMenu data provider” on page 852.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMxml.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">

    <fx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void
            {
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;
            }

            // Method to show the menu.
            private function show():void
            {
                myMenu.display(this.stage, 10, 10);
            }
        ]]>
    </fx:Script>

     <fx:Declarations>
        <!-- The XML data provider -->
        <fx:XML format="e4x" id="menuData">
            <root>
                <menuitem label="MenuItem A"/>
                <menuitem label="MenuItem B" type="check" toggled="true"/>
                <menuitem label="MenuItem C" enabled="false"/>
                <menuitem type="separator"/>
                <menuitem label="MenuItem D">
                    <menuitem label="SubMenuItem D-1"/>
                    <menuitem label="SubMenuItem D-2"/>
                    <menuitem label="SubMenuItem D-3"/>
                </menuitem>
            </root>
        </fx:XML>
     </fx:Declarations>

    <!-- Button control to create and open the menu. -->
    <s:Button x="300" y="10"
        label="Open Menu"
        click="show();"/>
</s:WindowedApplication>
```

Specifying menu item keyboard equivalents

You can specify a key equivalent (sometimes called an accelerator) for a menu command. When the key or key combination is pressed the FlexNativeMenu dispatches an `itemClick` event, as though the user had selected the menu item. The key equivalent string is automatically displayed beside the menu item name in the menu. The format depends on the user's operating system and system preferences. In order for the command to be invoked, the menu containing the command must be part of the application menu (OS X) or the window menu of the active window (Windows).

***Note:** Key equivalents are only triggered for application and window menus. If you add a key equivalent to a context or pop-up menu, the key equivalent is displayed in the menu label but the associated menu command is never invoked.*

About key equivalents

A key equivalent consists of two parts:

Primary key A string containing the character that serves as the key equivalent. If a data provider object has a `keyEquivalent` field, the DefaultDataDescriptor automatically uses that value as the key equivalent. You can specify an alternative data provider field by setting the FlexNativeMenu component's `keyEquivalentField` property. You can specify a function to use to determine the key equivalent by setting the FlexNativeMenu component's `keyEquivalentFunction` property.

Modifier keys One or more modifier keys that are also part of the key equivalent combination, such as the control key, shift key, command key, and so forth. If a data provider item includes an `altKey`, `commandKey`, `controlKey`, or `shiftKey` object field or XML attribute set to `true`, the specified key or keys become part of the key equivalent combination, and the entire key combination must be pressed to trigger the command. Alternatively, you can specify a function for the FlexNativeMenu component's `keyEquivalentModifiersFunction`, and that function is called to determine the key equivalent modifiers for each data provider item.

If you specify more than one key equivalent modifier, all the specified modifiers must be pressed in order to trigger the command. For instance, for the menu item generated from the following XML the key equivalent combination is Control-Shift-A (rather than Control-A OR Shift-A):

```
<menuitem label="Select All" keyEquivalent="a" controlKey="true" shiftKey="true"/>
```

Note that this can result in impossible key combinations if the menu item specifies multiple modifiers that are only available on one operating system. For example, the following item results in a menu item with the key equivalent Command-Shift-G:

```
<menuitem label="Ungroup" keyEquivalent="g" commandKey="true" shiftKey="true"/>
```

On Mac OS X, this command works as expected. On Windows, the key equivalent Shift-G is displayed in the menu. However, pressing Shift-G does not trigger the command because the Command key is still considered a required part of the command, even though that key doesn't exist in Windows.

To use different key combinations for the same menu item on different platforms, you can specify a `keyEquivalentModifiersFunction` function for the FlexNativeMenu instance. This function can provide alternative logic for processing the menu item data. For an example using the `keyEquivalentModifiersFunction`, see "Example: Using custom logic for multi-platform key equivalent menu commands" on page 855.

Example: FlexNativeMenu key equivalent commands

The following example uses a FlexNativeMenu to define a menu that includes keyboard equivalents for the menu commands. Note that while this example only uses keys and modifier keys that are available on Windows and Mac OS X, it uses the Control key as a modifier on both platforms. However, the Command key would be more in line with the common convention on Mac OS X. For an example that uses the `keyEquivalentModifiersFunction` property to create menus that use the common cross-platform conventions, see "Example: Using custom logic for multi-platform key equivalent menu commands" on page 855.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMKeyEquiv.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:menu>
        <mx:FlexNativeMenu dataProvider="{menuData}"
            labelField="@label"
            keyEquivalentField="@keyEquivalent"
            showRoot="false"
            itemClick="trace('click:', event.label);"/>
    </s:menu>
    <fx:Declarations>
        <fx:XML format="e4x" id="menuData">
            <root>
                <menuitem label="File">
                    <menuitem label="New" keyEquivalent="n" controlKey="true"/>
                    <menuitem label="Open" keyEquivalent="o" controlKey="true"/>
                    <menuitem label="Save" keyEquivalent="s" controlKey="true"/>
                    <menuitem label="Save As..." keyEquivalent="s" controlKey="true"
shiftKey="true"/>
                    <menuitem label="Close" keyEquivalent="w" controlKey="true"/>
                </menuitem>
                <menuitem label="Edit">
                    <menuitem label="Cut" keyEquivalent="x" controlKey="true"/>
                    <menuitem label="Copy" keyEquivalent="c" controlKey="true"/>
                    <menuitem label="Paste" keyEquivalent="v" controlKey="true"/>
                </menuitem>
            </root>
        </fx:XML>
    </fx:Declarations>
</s:WindowedApplication>
```

Example: Using custom logic for multi-platform key equivalent menu commands

The following example creates the same menu structure as the previous example. However, instead of using the same keyboard combination (for example, Control-O) regardless of the user's operating system, in this example a `keyEquivalentModifiersFunction` function is defined for the FlexNativeMenu. The function is used to create keyboard equivalents that follow platform conventions by using the Control key on Windows but substituting the Command key on Mac OS X.

In the data provider data, the `controlKey="true"` attribute is still used. The function that determines the key equivalent modifiers uses the value of the `controlKey` field or XML attribute to specify the Control key on Windows and the Command key on OS X, and if the `controlKey` attribute is `false` (or not specified) then neither modifier is applied.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMKeyEquivMultiP.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="init();">
    <s:menu>
        <mx:FlexNativeMenu dataProvider="{menuData}"
            labelField="@label"
            keyEquivalentField="@keyEquivalent"
            keyEquivalentModifiersFunction="keyEquivalentModifiers"
            showRoot="false"
            itemClick="trace('click:', event.label);"/>
    </s:menu>

    <fx:Script>
        <![CDATA[
            import flash.system.Capabilities;
            import flash.ui.Keyboard;

            private var isWin:Boolean;
            private var isMac:Boolean;

            private function init():void {
                isWin = (Capabilities.os.indexOf("Windows") >= 0);
                isMac = (Capabilities.os.indexOf("Mac OS") >= 0);
            }

            private function keyEquivalentModifiers(item:Object):Array {
                var result:Array = new Array();

                var keyEquivField:String = menu.keyEquivalentField;
                var altKeyField:String;
                var controlKeyField:String;
                var shiftKeyField:String;
                if (item is XML)
                {
                    altKeyField = "@altKey";
                    controlKeyField = "@controlKey";
                    shiftKeyField = "@shiftKey";
                }
                else if (item is Object)
                {
                    altKeyField = "altKey";
                    controlKeyField = "controlKey";
                    shiftKeyField = "shiftKey";
                }

                if (item[keyEquivField] == null || item[keyEquivField].length == 0)
                {
                    return result;
                }

                if (item[altKeyField] != null && item[altKeyField] == true)
                {
                    if (isWin)
                    {
                        result.push(Keyboard.ALTERNATE);
                    }
                }

                if (item[controlKeyField] != null && item[controlKeyField] == true)
                {
                    if (isWin)
                    {
                        result.push(Keyboard.CONTROL);
                    }
                    else if (isMac)
                    {
                        result.push(Keyboard.COMMAND);
                    }
                }

                if (item[shiftKeyField] != null && item[shiftKeyField] == true)
                {
                    result.push(Keyboard.SHIFT);
                }

                return result;
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:XML format="e4x" id="menuData">
            <root>
                <menuitem label="File">
                    <menuitem label="New" keyEquivalent="n" controlKey="true"/>
                    <menuitem label="Open" keyEquivalent="o" controlKey="true"/>
                    <menuitem label="Save" keyEquivalent="s" controlKey="true"/>
                    <menuitem label="Save As..."
                        keyEquivalent="s"
                        controlKey="true"
                        shiftKey="true"/>
                    <menuitem label="Close" keyEquivalent="w" controlKey="true"/>
                </menuitem>
                <menuitem label="Edit">
                    <menuitem label="Cut" keyEquivalent="x" controlKey="true"/>
                    <menuitem label="Copy" keyEquivalent="c" controlKey="true"/>
                    <menuitem label="Paste" keyEquivalent="v" controlKey="true"/>
                </menuitem>
            </root>
        </fx:XML>
    </fx:Declarations>
</s:WindowedApplication>
```

Specifying menu item mnemonics

A menu item mnemonic is a key associated with a menu item which, when pressed while the menu is displayed, is equivalent to selecting that menu item with the mouse. Typically, the operating system indicates a menu item’s mnemonic by underlining that character in the name of the menu item. Mnemonics for menu items are supported in Windows. In Mac OS X, when a menu is activated a user types the first letter or letters of a menu item’s label, then presses return to select the item.

The simplest way to specify a mnemonic for a menu item in a FlexNativeMenu component is to include an underscore character (“_”) in the menu item’s label field, immediately to the left of the letter that serves as the mnemonic for that menu item. For instance, if the following XML node is used in a data provider for a FlexNativeMenu, the mnemonic for the command is the first character of the second word (the letter “A”):

```
<menuitem label="Save _As"/>
```

When the native menu is created, the underscore character is not included in the label. Instead, the character following the underscore becomes the mnemonic for the menu item. To include a literal underscore character in a menu item’s name, use two underscore characters (“__”). This sequence is converted to one underscore in the menu item label.

As an alternative to using underscore characters in label names, you can provide an integer index position for the mnemonic character in a `mnemonicIndex` field in the data provider objects. You can also use another Object property or XML attribute by setting the FlexNativeMenu component’s `mnemonicIndexField` property. To use complex logic for assigning mnemonics, you can specify a function for the FlexNativeMenu component’s `mnemonicIndexFunction` property. Each of these properties provides a mechanism to define an integer (zero-based) index position for the menu items’ mnemonics.

Handling FlexNativeMenu control events

User interaction with a FlexNativeMenu is event-driven. When the user selects a menu item or opens a menu or submenu, the menu dispatches an event. You can register event listeners to define the actions that are carried out in response to the user’s selection. Event handling with the FlexNativeMenu component shares similarities with other Flex menu components, but also has key differences. For information about Flex menu component events, see Menu-based control events.

The FlexNativeMenu component defines two specific events, both of which dispatch event objects that are instances of the FlexNativeMenuEvent class:

Event	Description
`itemClick`	(FlexNativeMenuEvent.ITEM_CLICK) Dispatched when a user selects an enabled menu item of type normal or check. This event is not dispatched when a user selects a menu item that opens a submenu or a disabled menu item.
`menuShow`	(FlexNativeMenuEvent.MENU_SHOW) Dispatched when the entire menu or a submenu opens (including a top-level menu of an application or window menu).

The event object passed to the event listener is of type FlexNativeMenuEvent and contains the following menu-specific properties:

Property	Description
`index`	The index of the item in the menu or submenu that contains it. Only available for the `itemClick` event.
`item`	The item in the data provider for the menu item associated with the event. Only available for the `itemClick` event.
`label`	The label of the item. Only available for the `itemClick` event.
`nativeMenu`	A reference to the underlying NativeMenu object where the event occurred.
`nativeMenuItem`	A reference to the underlying NativeMenuItem object that triggered the event. Only available for the `itemClick` event.

To access properties of an object-based menu item, you specify the `item` property of the event object, as follows:

```
ta1.text = event.item.extraData;
```

To access attributes of an E4X XML-based menu item, you specify the menu item attribute name in E4X syntax, as follows:

```
ta1.text = event.item.@extraData;
```

***Note:** If you set an event listener on a submenu of a FlexNativeMenu component, and the menu data provider's structure changes (for example, if an element is removed), the event listener might no longer exist. To ensure that the event listener is available when the data provider structure changes, use the events of the FlexNativeMenu control, not a submenu.*

The standard approach to handling FlexNativeMenu events is to register an event listener with the FlexNativeMenu component. Any time an individual menu item is selected or submenu is opened, the FlexNativeMenu dispatches the appropriate event. Your listener code can use the event object's `item` property or other properties to determine on which menu item the interaction occurred, and perform actions in response.

The following example lets you experiment with FlexNativeMenu control events. It lets you display two menus, one with an XML data provider and one with an Array data provider. A TextArea control displays information about each event as a user opens the menus, opens submenus, and selects menu items. The example shows some of the differences in how you handle XML and object-based menus. It also indicates some of the types of information that are available about each FlexNativeMenu event.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/NativeMEvents.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.events.FlexNativeMenuEvent;
            import mx.controls.FlexNativeMenu;
            import mx.events.FlexNativeMenuEvent;

            // The event listener that opens the menu with an XML data
            // provider and adds event listeners for the menu.
            private function createAndShow():void {
                ta1.text="";
               xmlBasedMenu.addEventListener(FlexNativeMenuEvent.ITEM_CLICK, menuShowInfo);
                xmlBasedMenu.addEventListener(FlexNativeMenuEvent.MENU_SHOW, menuShowInfo);
                xmlBasedMenu.display(stage, 225, 10);
            }
            // The event listener for the xml-based menu events.
            // Retain information on all events for a menu instance.
            private function menuShowInfo(event:FlexNativeMenuEvent):void {
                ta1.text = "event.type: " + event.type;

                // The label field is null for menuShow events.
                ta1.text += "\nevent.label: " + event.label;

                // The index value is -1 for menuShow events.
                ta1.text+="\nevent.index: " + event.index;

                // The item field is null for menuShow events.
                if (event.item != null)
                {
                    ta1.text += "\nItem label: " + event.item.@label
                    ta1.text += "\nItem toggled: " + event.item.@toggled;
                    ta1.text += "\nItem type: " + event.item.@type;
                }
            }
            // The event listener that creates an object-based menu
            // and adds event listeners for the menu.
            private function createAndShow2():void {
                ta1.text="";
            objectBasedMenu.addEventListener(FlexNativeMenuEvent.ITEM_CLICK, menuShowInfo2);
             objectBasedMenu.addEventListener(FlexNativeMenuEvent.MENU_SHOW, menuShowInfo2);
                objectBasedMenu.display(stage, 225, 10);
            }
            // The event listener for the object-based Menu events.
            private function menuShowInfo2(event:FlexNativeMenuEvent):void {
                ta1.text = "event.type: " + event.type;

                // The label field is null for menuShow events.
                ta1.text += "\nevent.label: " + event.label;

                // The index value is -1 for menuShow events.
                ta1.text += "\nevent.index: " + event.index;

                // The item field is null for menuShow events.
                if (event.item)
                {
                    ta1.text += "\nItem label: " + event.item.label
                    ta1.text += "\nItem toggled: " + event.item.toggled;
                    ta1.text += "\ntype: " + event.item.type;
                }
            }
            // The object-based data provider, an Array of objects.
            // Its contents are identical to that of the XML data provider.
            public var objMenuData:Array = [
                {label: "MenuItem A", children: [
                    {label: "SubMenuItem A-1", enabled: false},
                    {label: "SubMenuItem A-2"}
                ]},
                {label: "MenuItem B", type: "check", toggled: true},
                {label: "MenuItem C", type: "check", toggled: false},
                {type: "separator"},
                {label: "MenuItem D", children: [
                    {label: "SubMenuItem D-1"},
                    {label: "SubMenuItem D-2"},
                    {label: "SubMenuItem D-3"}
                ]}
            ];
        ]]>
    </fx:Script>

    <fx:Declarations>
        <!-- The XML-based menu data provider.
        The <fx:XML tag requires a single root. -->
        <fx:XML id="xmlMenuData" format="e4x">
            <xmlRoot>
                <menuitem label="MenuItem A" >
                    <menuitem label="SubMenuItem A-1" enabled="false"/>
                    <menuitem label="SubMenuItem A-2"/>
                </menuitem>
                <menuitem label="MenuItem B" type="check" toggled="true"/>
```

```
                <menuitem label="MenuItem C" type="check" toggled="false"/>
                <menuitem type="separator"/>
                <menuitem label="MenuItem D">
                    <menuitem label="SubMenuItem D-1"/>
                    <menuitem label="SubMenuItem D-2"/>
                    <menuitem label="SubMenuItem D-3"/>
                </menuitem>
            </xmlRoot>
        </fx:XML>
        <!-- Define the FlexNativeMenu components in
          the <fx:Declarations> block. -->
         <mx:FlexNativeMenu id="xmlBasedMenu"
            showRoot="false"
            labelField="@label"
            dataProvider="{xmlMenuData}"/>
        <mx:FlexNativeMenu id="objectBasedMenu"
            dataProvider="{objMenuData}"/>
    </fx:Declarations>

    <!-- Button controls to open the menus. -->
    <s:Button x="10" y="5"
        label="Open XML Popup"
        click="createAndShow();"/>
    <s:Button x="10" y="35"
        label="Open Object Popup"
        click="createAndShow2();"/>
    <!-- Text area to display the event information -->
    <s:TextArea x="10" y="70"
        width="200" height="250"
        id="ta1"/>
</s:WindowedApplication>
```

About the AIR window containers

Flex containers define the content, sizing, and positioning for a specific part of an application. For AIR applications, Flex includes two specific window components that serve as containers whose content area is an operating system window. Both the WindowedApplication and the Window containers can be used to define the contents of an operating system window. They also provide the means to define and control characteristics of the window itself, such as the window's size, its position on the user's screen, and the presence of window chrome.

Spark and MX window containers

Flex supplies a Spark and a MX version of the WindowedApplication and the Window containers. The Spark components are located in the spark.components package. The MX components are located in the mx.core package.

The biggest difference between the two component sets is that you use skins to set many of the visual characteristics of the Spark components, and styles to set them for the MX components. Adobe recommends that you use the Spark components in your application when possible.

WindowedApplication container

The WindowedApplication container defines an application container that you use to create Flex applications for AIR that use the native operating system chrome. The WindowedApplication container adds window-related functionality and desktop application-specific functionality to the Flex Application container, which you can use when you build AIR applications.

The WindowedApplication container serves two roles. First, it provides the entry point into the main application, which in turn executes other application logic. In that sense it serves as the core of the entire application, just as the Application container does for a browser-based Flex application. Second, the WindowedApplication container represents the first native window of the application.

If the application only uses one native window, the WindowedApplication is the base stage that contains all other content. If your application opens additional native windows, each window has its own stage and display list.

The native window defined by the WindowedApplication is no different from any other application window in this respect. This is different from a browser-based Flex application, where all of an application's windows are drawn by Flex within the same stage (the Application container).

For example, in a Flex AIR application, registering a `keyDown` event listener on the WindowedApplication container only dispatches events when a key is pressed while the initial window has focus. If the application has multiple native windows and another of the windows has focus when the key is pressed, the event is not dispatched. This behavior differs from a non-AIR Flex application, in which a `keyDown` listener registered with the Application container receives notification of all key presses while the application has focus.

For more information on the WindowedApplication container, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating and using a WindowedApplication container

The WindowedApplication container defines an AIR application object that includes its own window controls. In an MXML AIR application, a Spark `<s:WindowedApplication>` or MX `<mx:WindowedApplication>` tag replaces the Application tag in a Flex application.

WindowedApplication container example

The following application shows a simple use of the WindowedApplication container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/WindowSimple.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

       <s:Label text="Hello World" />
</s:WindowedApplication>
```

Window container

The Window container is a Flex container that is used to define the content and layout of operating system windows that are opened after an application launches. In other words, it is used for windows other than the initial or main window of the application, which is a WindowedApplication container.

In addition to the functionality that the Window container shares with the WindowedApplication container, a Window container allows you to define the main characteristics of the window. The characteristics you can specify include the type of window, the type of chrome, whether certain actions (such as resizing and maximizing) are permitted for the window, and more.

These characteristics are accessed as properties that can be set when the container is initially created, before the actual operating system window is displayed. However, once the actual window is opened, the properties can no longer be set and can only be read.

For more information about the Window container, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating and using a Window container

The Window container defines an AIR application object that includes its own window controls. In an MXML AIR application, you use a `<s:Window>` tag as the top-level tag of an MXML component, with the window's content defined in the body of the MXML component document. However, unlike other MXML components, a Window-based component cannot be used in another MXML document. Instead, you create an instance of the MXML component in ActionScript.

Because several of the properties of the Window container can only be set before the window is opened, they can be set as properties in the `<s:Window>` MXML tag. They can also be set using ActionScript, either in an `<fx:Script>` block in the window's MXML document or in code that creates an instance of the window.

Once the window's initial properties are set, you call the Window container's `open()` method to cause the operating system window to appear on the user's display.

Window container example

The following example shows a basic use of the Window container. The example includes two MXML files. The first uses a WindowedApplication container and is the initial window of the application. The second uses the Window container to define a secondary window for the application. In this example, the main window simulates a "splash screen" for the application. After a set time (4 seconds) it closes the splash screen and opens the second window. In order to make a splash screen window with no window chrome, in the application.xml file the `systemChrome` tag is set to `none`.

The following code defines the main application MXML file, which contains the initial window (the splash screen) that opens automatically when the application is run:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/WindowSplash.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">

    <fx:Script>
        <![CDATA[
            import myComponents.DocumentWindow;
            private const LOAD_DELAY:int = 4;
            private var timeElapsed:int = 0;
            private var loadTimer:Timer;

            private var docWindow:myComponents.DocumentWindow;

            private function init():void {
                // center the window on the screen
                var screenBounds:Rectangle = Screen.mainScreen.bounds;
                nativeWindow.x = (screenBounds.width - nativeWindow.width) / 2;
                nativeWindow.y = (screenBounds.height - nativeWindow.height) / 2;

                // start the timer, which simulates a loading delay
                loadTimer = new Timer(1000);
                loadTimer.addEventListener(TimerEvent.TIMER, incrementTime);
                loadTimer.start();

                updateStatus();
            }
            private function incrementTime(event:TimerEvent):void {
                timeElapsed++;

                updateStatus();

                // if the loading delay has passed, stop the timer,
                // close the splash screen, and open the document window
                if ((LOAD_DELAY - timeElapsed) == 0)
                {
                    loadTimer.stop();
                    loadTimer.removeEventListener(TimerEvent.TIMER, incrementTime);
                    loadTimer = null;

                    nativeWindow.close();

                    // open a new instance of the document window
                    docWindow = new DocumentWindow();
                    docWindow.open();
                }
            }

            private function updateStatus():void {
                var timeRemaining:uint = LOAD_DELAY - timeElapsed;
                var timeRemainingMsg:String = timeRemaining.toString() + " second";
                if (timeRemaining != 1) { timeRemainingMsg += "s"; }
                timeRemainingMsg += " remaining.";

                loadStatusMessage.text = "initializing... " + timeRemainingMsg;
            }
        ]]>
    </fx:Script>

    <s:VGroup horizontalCenter="0" verticalCenter="0">
        <s:Label text="My Splash Screen"
            fontFamily="Courier New"
            fontSize="36"/>
        <s:Label id="loadStatusMessage"
            text="initializing..."/>
    </s:VGroup>
</s:WindowedApplication>
```

The `incrementTime()` method is called each second and when the appropriate time is reached, a DocumentWindow instance is created and its `open()` method is called. The DocumentWindow class is defined in a separate MXML document. Its base MXML tag is the `<s:Window>` tag, so it is a subclass of the Window class (the Window container). Here is the source code for the DocumentWindow MXML file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/myComponents/DocumentWindow.mxml -->
<s:Window xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="550" height="450">
    <s:Label text="This is a document window."
        horizontalCenter="0" verticalCenter="0"/>
</s:Window>
```

Controlling window chrome

The window that a WindowedApplication or Window container defines conforms to the standard behavior of the operating system. The user can move the window by dragging the title bar and resize the window by dragging on any side or corner of the window. The containers also include a set of properties that allow you to control window sizing, including `minimumHeight`, `minimumWidth`, `maximumHeight`, and `maximumWidth`.

The WindowedApplication container and the Window container allow you to control the presence and the appearance of the window *chrome*.

For the MX WindowedApplication or Window containers only, you can hide the chrome by setting the `showFlexChrome` style to `false`.

Setting the chrome

Flex provides several options for setting the chrome:

- Use the chrome defined by your operating system. The system chrome consists of a border and title bar. The title bar contains the close, minimize, maximize, and restore buttons. For an Apple computer, the restore button is omitted.
- Use the Flex chrome. The Flex chrome is defined by a set of skin files. The Flex chrome contains all of the elements of the system chrome, and adds a gripper to the lower-right corner of the container that you can use to resize the window. The Flex chrome also adds a status bar to the bottom of the window.
- Use the WindowedApplicationSkin class. The WindowedApplicationSkin skin class contains only a border and a status bar.
- Create your own skin files to define the chrome.

The title bar area of the chrome includes a title message and an icon that can be set and modified by using the `title` and `titleIcon` properties of the WindowedApplication or Window container. You can show or hide the status bar by using the `showStatusBar` property.

How the chrome is represented depends on the setting of the `systemChrome` attribute, which is defined in the application .xml file for a WindowedApplication container, or in the Window container's `systemChrome` property. If `systemChrome` is set to "standard" in the application .xml file (or `flash.display.NativeWindowSystemChrome.STANDARD` in ActionScript) the operating system renders the chrome.

If `systemChrome` is set to "none" (`NativeWindowSystemChrome.NONE`) the Window and WindowedApplication container can either use the Flex chrome, use the WindowedApplicationSkin class to define the chrome, or define no chrome at all. When you do not use any chrome, the application shows only a background color.

The Flex chrome is defined by the SparkChromeWindowedApplicationSkin skin class in the spark.skins.spark package. The following example sets the `skinClass` style of the WindowedApplication to specify to use the Flex chrome. This example assumes that you set `systemChrome` to "none" in the application's .xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/WindowSimpleChrome.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Style>
        @namespace "library://ns.adobe.com/flex/spark";
        WindowedApplication
        {
            skinClass:ClassReference("spark.skins.spark.SparkChromeWindowedApplicationSkin");
        }
    </fx:Style>

    <s:Label text="Hello World" />
</s:WindowedApplication>
```

Flex includes a simple skin class for the WindowedApplication and Window containers that defines just a border and status bar, but no gripper button, title bar, or title bar buttons. The following example assigns this skin to the WindowedApplication container. This example assumes that you set `systemChrome` to “none” in the application’s .xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- aircomponents/src/WindowSimpleChromeWindowed.mxml -->
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Style>
        @namespace "library://ns.adobe.com/flex/spark";
        WindowedApplication
        {
            skinClass:ClassReference("spark.skins.spark.WindowedApplicationSkin");
        }
    </fx:Style>

    <s:Label text="Hello World" />
</s:WindowedApplication>
```

Defining custom skin classes

The SparkChromeWindowedApplicationSkin class references additional skin classes to draw the gripper, title bar, and title bar buttons. Different operating systems can use the same skin file for some skins, but use different skin files for other skins. The following table describes these classes:

Skin	Microsoft Window skin file	Apple Macintosh skin file
The skin for the close button in the title bar.	CloseButtonSkin.mxml	MacCloseButtonSkin.mxml
The skin for the gripper button for resizing the window.	GripperSkin.mxml	GripperSkin.mxml
The skin for the maximize button in the title bar.	MaximizeButtonSkin.mxml	MacMaximizeButtonSkin.mxml
The skin for the minimize button in the title bar.	MinimizeButtonSkin.mxml	MacMinimizeButtonSkin.mxml
The title bar skin.	TitleBarSkin.mxml	MacTitleBarSkin.mxml
The skin for the restore button in the title bar.	RestoreButtonSkin.mxml	No restore button available.

To customize these skins, define your own skin class based on the SparkChromeWindowedApplicationSkin class, and then reference your custom skin classes from it.

Dynamically repeating controls and containers

Repeater components let you dynamically repeat MXML content at run time.

Adobe® Flex® also supports list controls, which provide better performance when you display large amounts of data. For more information about the HorizontalList, TileList, and List controls, see "MX data-driven controls" on page 943.

About Repeater components

Repeater components are useful for repeating a small set of simple user interface components, such as RadioButton controls and other controls typically used in Form containers. Repetition is generally controlled by an array of dynamic data, such as an Array object returned from a web service, but you can use static arrays to emulate simple `for` loops.

Although Repeater components look like containers in your code, they are not containers and have none of the automatic layout functionality of containers. Their sole purpose is to specify a series of subcomponents to include in your application one or more times based on the contents of a specified data provider. To align items that a repeater generates or perform any other layout task, place the Repeater component and its contents inside a container and apply the layout to that container.

Using the Repeater component

You use the `<mx:Repeater>` tag to declare a Repeater component that handles repetition of one or more user interface components based on dynamic or static data arrays at run time. The repeated components can be controls or containers. Using a Repeater component requires data binding to allow for run-time-specific values. For more information about data binding, see "Storing data" on page 889.

You can use the `<mx:Repeater>` tag anywhere a control or container tag is allowed, as long as you wrap the repeater tag in a MX container, such as VBox, HBox, Panel, or Canvas. If you use a container such as Canvas that uses absolute positioning, you must manually position each repeated element so that they do not overlap.

To repeat user interface components, you place their tags within the `<mx:Repeater>` tag. All components derived from the UIComponent class can be repeated with the exception of the `Application` container tag. Components that are not based on UIComponent class cannot be children of the Repeater control.

You can also use more than one `<mx:Repeater>` tag in an MXML document, and you can nest `<mx:Repeater>` tags.

Declaring the Repeater component in MXML

You declare the Repeater component in the `<mx:Repeater>` tag, inside a MX container. The following table describes the Repeater component's properties:

Property	**Description**
`id`	Instance name of the corresponding Repeater component.
`dataProvider`	An implementation of the ICollectionView interface, IList interface, or Array class, such as an ArrayCollection object. You must specify a `dataProvider` value or the Repeater component will not execute. Generally, you specify the value of the `dataProvider` property as a binding expression because the value is not known until run time.
`startingIndex`	Number that specifies the element in the data provider at which the repetition starts. The data provider array is zero-based, so to start at the second element of the array, specify a starting index of one. If the `startingIndex` is not within the range of the `dataProvider` property, no repetition occurs.
`count`	Number that specifies how many repetitions occur. If the `dataProvider` property has fewer items than the number in the `count` property, the repetition stops with the last item.
`currentIndex`	Number that specifies the element of the `dataProvider` item currently being processed. The data provider array is zero-based, so when the third element is being processed, the current index is two. This property changes as the Repeater component executes, and is -1 after the execution is complete. It is a read-only property that you cannot set in the `<mx:Repeater>` tag.
`currentItem`	Reference to the item that is being processed in the `dataProvider` property. This property changes as the Repeater component executes, and is null after the execution is complete. It is a read-only property that you cannot set in the `<mx:Repeater>` tag. After a Repeater component finishes repeating, you do not use the `currentItem` property to get the current item. Instead, you call the `getRepeaterItem()` method of the repeated component itself. For more information, see "Referencing repeated components" on page 873.
`recycleChildren`	Boolean value that, when set to `true`, binds new data items into existing Repeater children, incrementally creates new children if there are more data items, and destroys extra children that are no longer required. For more information, see "How a Repeater component executes" on page 883.

The following table describes the Repeater component's events:

Event	**Description**
`repeat`	Dispatched each time an item is processed and `currentIndex` and `currentItem` are updated.
`repeatEnd`	Dispatched after all the subcomponents of a repeater are created.
`repeatStart`	Dispatched when Flex begins processing the `dataProvider` property and begins creating the specified subcomponents.

Basic principles of the Repeater component

The simplest repeating structures you can create with a Repeater component are static loops that execute a set number of times. The following Repeater component emulates a simple for loop that executes four times, printing a simple line of text and incrementing the counter by one each time:

```
<?xml version="1.0"?>
<!-- repeater/StaticLoop.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns="*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var myArray:Array=[1,2,3,4];
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ArrayCollection id="myAC" source="{myArray}"/>
    </fx:Declarations>
    <!-- Notice that the Repeater is inside a MX container. -->
    <mx:VBox>
        <mx:Repeater id="myrep" dataProvider="{myAC}">
            <mx:Label id="Label1" text="This is loop #{myrep.currentItem}"/>
        </mx:Repeater>
    </mx:VBox>

</s:Application>
```

In actuality, the counter is not the data within the array, but the position within the array. As long as the array contains four elements, you can provide any data within the array itself, and the Repeater component still executes four times. The following example illustrates this principle:

```
<?xml version="1.0"?>
<!-- repeater/StaticLoop2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var myArray:Array=[10,20,30,40];
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ArrayCollection id="myAC" source="{myArray}"/>
    </fx:Declarations>
    <!-- Notice that the Repeater is inside a MX container. -->
    <mx:VBox>
        <mx:Repeater id="myrep" dataProvider="{myAC}">
            <mx:Label id="Label1" text="This is loop #{myrep.currentIndex+1}"/>
        </mx:Repeater>
    </mx:VBox>

</s:Application>
```

Notice that this example prints the current index plus one, not the index itself. This is because the first element of the array is assigned an index value of zero (0).

You can also use the `startingIndex` and `count` properties to adjust the starting point and total number of executions. The `count` property sets a maximum on the number of times the Repeater component executes. The `count` property is often—but not always—an exact measure of the number of times that the content within the Repeater component will execute. This number is because the Repeater component stops after the last element of the data provider is reached, regardless of the value of the `count` property.

The following example shows how the `count` and `startingIndex` properties affect a Repeater component:

```
<?xml version="1.0"?>
<!-- repeater\StartingIndexCount.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var myArray:Array=[100,200,300,400,500,600];
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ArrayCollection id="myAC" source="{myArray}"/>
    </fx:Declarations>
    <mx:VBox>
        <mx:Repeater id="myrep" dataProvider="{myAC}" count="6">
            <mx:Label id="Label1"
                text="Value: {myrep.currentItem}, Loop #{myrep.currentIndex+1} of
{myrep.count}"/>
        </mx:Repeater>
    </mx:VBox>
    <mx:HRule/>
    <mx:VBox>
        <mx:Repeater id="myrep2" dataProvider="{myAC}"
                count="4" startingIndex="1">
            <mx:Label id="Label2"
                text="Value: {myrep2.currentItem}, Loop #{myrep2.currentIndex-
myrep2.startingIndex+1} of {myrep2.count}"/>
        </mx:Repeater>
    </mx:VBox>
    <mx:HRule/>
    <mx:VBox>
        <mx:Repeater id="myrep3" dataProvider="{myAC}" count="6"
                startingIndex="3">
            <mx:Label id="Label3"
                text="Value: {myrep3.currentItem}, Loop #{myrep3.currentIndex-
myrep3.startingIndex+1} of {myrep3.count}"/>
        </mx:Repeater>
    </mx:VBox>
</s:Application>
```

The first Repeater component loops through each element of the data provider, starting with the first and stopping after the last. The second Repeater component starts at the second element of the data provider and iterates four times, ending at the fifth element. The third Repeater component starts with the fourth element and continues until the end of the data provider array, and then stops. Only three iterations occur despite the fact that the `count` property is set to 6.

Creating dynamic loops with the Repeater component

Instead of a static data written directly into your application, your application might work with dynamic data providers. Dynamic data is defined in an `<fx:Model>` tag and drawn from an XML file, a web service, a remote object, or some other source. The data is collected and evaluated at run time to determine the number and value of elements in the data provider and how the Repeater component behaves.

In the following example, an `<mx:Repeater>` tag repeats a RadioButton control for each product in an XML file:

```
<?xml version="1.0"?>
<!-- repeater\DynamicLoop.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="catalogService.send();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:HTTPService id="catalogService" url="assets/repeater/catalog.xml"
                resultFormat="e4x"/>
        <mx:XMLListCollection id="myXC"
                source="{catalogService.lastResult.product}"/>
    </fx:Declarations>
    <mx:VBox>
        <mx:Repeater id="r" dataProvider="{myXC}">
            <mx:RadioButton id="Radio" label="{r.currentItem.name}"
                    width="150"/>
        </mx:Repeater>
    </mx:VBox>
</s:Application>
```

Assume that catalog.xml contains the following:

```
<?xml version="1.0"?>
<!-- assets\catalog.xml -->
<products>
  <product>
    <name>Name</name>
    <price>Price</price>
    <freeship>Free Shipping?</freeship>
  </product>
  <product>
    <name>Whirlygig</name>
    <price>5</price>
    <freeship>false</freeship>
  </product>
  <product>
    <name>Tilty Thingy</name>
    <price>15</price>
    <freeship>true</freeship>
  </product>
<product>
    <name>Really Big Blocks</name>
    <price>25</price>
    <freeship>true</freeship>
  </product>
</products>
```

You can still use the `count` property to restrict the number of iterations performed. You can use the `startingIndex` property to skip entries at the beginning of the data provider.

In the preceding example, the first product entry in the XML file contains metadata that other applications use to create column headers. Because it doesn't make sense to include that entry in the radio buttons, start the Repeater component at the second element of the data provider, as in the following code example:

```
<?xml version="1.0"?>
<!-- repeater\StartSecondElement.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="catalogService.send();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:HTTPService id="catalogService" url="assets/repeater/catalog.xml"
                resultFormat="e4x"/>
        <mx:XMLListCollection id="myXC"
                source="{catalogService.lastResult.product}"/>
    </fx:Declarations>


    <mx:VBox>
        <mx:Repeater id="r" dataProvider="{myXC}"
                startingIndex="1">
            <mx:RadioButton id="Radio" label="{r.currentItem.name}"
                width="150"/>
        </mx:Repeater>
    </mx:VBox>
</s:Application>
```

Referencing repeated components

To reference individual instances of a repeated component, you use indexed `id` references, as the following example shows:

```
<?xml version="1.0"?>
<!-- repeater\RefRepeatedComponents.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function labelTrace():void {
                for (var i:int = 0; i < nameLabel.length; i++)
                    trace(nameLabel[i].text);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:Model id="catalog" source="../assets/repeater/catalog.xml"/>
        <s:ArrayCollection id="myAC" source="{catalog.product}"/>
    </fx:Declarations>

    <s:Label id="title" text="Products:"/>

    <s:VGroup>
        <mx:VBox>
            <mx:Repeater id="r" dataProvider="{myAC}" startingIndex="1">
                <mx:Label id="nameLabel"
                    text="{r.currentItem.name}: ${r.currentItem.price}"
                    width="200"/>
            </mx:Repeater>
        </mx:VBox>
        <s:Button label="Trace" click="labelTrace();"/>
    </s:VGroup>
</s:Application>
```

In this example, the `id` of the repeated Label control is `nameLabel`; each `nameLabel` instance created has this `id`. You reference the individual Label instances as `nameLabel[0]`, `nameLabel[1]`, and so on. You reference the total number of `nameLabel` instances as `nameLabel.length`. The `for` loop traces the `text` property of each Label control in the `nameLabel` Array object. The `labelTrace()` method prints the name and price of each product in the system log, provided you've defined it in the `mm.cfg` file.

Referencing repeated child components

When a container is repeated and indexed in an array, its children are also indexed. For example, for the following MXML code, you reference the child Label controls of the VBox container `vb[0]` as `nameLabel[0]` and `shipLabel[0]`. The syntax for referencing the children is the same as the syntax for referencing the parent.

```
<?xml version="1.0"?>
<!-- repeater\RefRepeatedChildComponents.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            public function labelTrace():void {
                for (var i:int = 0; i < nameLabel.length; i++)
                    trace(nameLabel[i].text);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:Model id="catalog" source="../assets/repeater/catalog.xml"/>
    </fx:Declarations>

    <s:Label id="title" text="Products:"/>
    <mx:VBox>
        <mx:Repeater id="r" dataProvider="{catalog.product}"
                startingIndex="1">
            <mx:VBox id="vb">
                <mx:Label id="nameLabel"
                    text="{r.currentItem.name}: ${r.currentItem.price}"
                    width="200"/>
                <mx:Label id="shipLabel"
                    text="Free shipping: {r.currentItem.freeship}"/>
                <mx:Spacer/>
            </mx:VBox>
        </mx:Repeater>
    </mx:VBox>
    <s:Button label="Trace" click="labelTrace();"/>
</s:Application>
```

Referencing nested Repeater components

When `<mx:Repeater>` tags are nested, the inner `<mx:Repeater>` tags are indexed Repeater components. For example, for the following MXML code, you access the nested Repeater components as `r2[0]`, `r2[1]`, and so on. The syntax for referencing the children is same as the syntax for referencing the parent.

```
<?xml version="1.0"?>
<!-- repeater\RefNestedComponents.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[

      public function labelTrace():void {
        for (var i:int = 0; i < nameLabel.length; i++)
         for (var j:int = 0; j < nameLabel[i].length; j++)
          trace(nameLabel[i][j].text);
      }
    ]]>
  </fx:Script>
  <fx:Declarations>
      <fx:Model id="data">
        <color>
          <colorName>Red</colorName>
          <colorName>Yellow</colorName>
          <colorName>Blue</colorName>
        </color>
      </fx:Model>
      <fx:Model id="catalog" source="../assets/repeater/catalog.xml"/>
      <s:ArrayCollection id="myAC1" source="{catalog.product}"/>
      <s:ArrayCollection id="myAC2" source="{data.colorName}"/>
  </fx:Declarations>
  <s:Label id="title" text="Products:"/>
  <mx:VBox>
      <mx:Repeater id="r" dataProvider="{myAC1}" startingIndex="1">
        <mx:Repeater id="r2" dataProvider="{myAC2}">
          <mx:Label id="nameLabel" text="{r2.currentItem} {r.currentItem.name}:
${r.currentItem.price}" width="250"/>
        </mx:Repeater>
      </mx:Repeater>
  </mx:VBox>
  <s:Button label="Trace" click="labelTrace();"/>
</s:Application>
```

This application places the full list of products with color and price into the system log when you click the Button control (provided you have defined the log file in the mm.cfg file).

In the previous example, the instances of the Label control are multiply indexed because they are inside multiple Repeater components. For example, the index `nameLabel[1][2]` contains a reference to the Label control produced by the second iteration of `r` and the third iteration of `r2`.

Using the getRepeaterItem() method in an event handler for the Repeater component

When a Repeater component is busy repeating, each repeated object that it creates can bind at that moment to the Repeater component's `currentItem` property, which is changing as the Repeater component repeats. You cannot give each instance its own event handler by writing something like `click="doSomething({r.currentItem})"` because binding expressions are not allowed in event handlers, and all instances of the repeated component must share the same event handler.

Repeated components and repeated Repeater components have a `getRepeaterItem()` method that returns the item in the `dataProvider` property that was used to produce the object. When the Repeater component finishes repeating, you can use the `getRepeaterItem()` method to determine what the event handler should do based on the `currentItem` property. To do so, you pass the `event.currentTarget.getRepeaterItem()` method to the event handler. The `getRepeaterItem()` method takes an optional index that specifies which Repeater components you want when nested Repeater components are present; the 0 index is the outermost Repeater component. If you do not specify the index argument, the innermost Repeater component is implied.

Note: *After a Repeater component finishes repeating, you do not use the* `Repeater.currentItem` *property to get the current item. Instead, you call the* `getRepeaterItem()` *method of the repeated component itself.*

The following example illustrates the `getRepeaterItem()` method. When the user clicks each repeated Button control, the corresponding `colorName` value from the data model appears in the Button control label.

```
<?xml version="1.0"?>
<!-- repeater\GetItem.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
      public function clicker(cName:String):void {
         foolabel.text=cName;
       }
    ]]>
  </fx:Script>
  <s:Label id="foolabel" text="foo"/>
  <fx:Declarations>
      <fx:Model id="data">
        <color>
          <colorName>Red</colorName>
          <colorName>Yellow</colorName>
          <colorName>Blue</colorName>
        </color>
      </fx:Model>
      <s:ArrayCollection id="myAC" source="{data.colorName}"/>
  </fx:Declarations>
  <mx:VBox>
      <mx:Repeater id="myrep" dataProvider="{myAC}">
        <mx:Button click="clicker(event.currentTarget.getRepeaterItem());"
          label="{myrep.currentItem}"/>
      </mx:Repeater>
  </mx:VBox>
</s:Application>
```

The code in the following example uses the `getRepeaterItem()` method to display a specific URL for each Button control that the user clicks. The Button controls must share a common data-driven click handler, because you cannot use binding expressions inside event handlers. However, the `getRepeaterItem()` method lets you change the functionality for each Button control.

```
<?xml version="1.0"?>
<!-- repeater\DisplayURL.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
      [Bindable]
      public var dp:Array = [
        { label: "Flex", url: "http://www.adobe.com/flex" },
        { label: "Flash", url: "http://www.adobe.com/flash" }
      ];
    ]]>
  </fx:Script>
  <fx:Declarations>
      <s:ArrayCollection id="myAC" source="{dp}"/>
  </fx:Declarations>
  <mx:VBox>
      <mx:Repeater id="r" dataProvider="{myAC}">
        <mx:Button label="{r.currentItem.label}"
           click="navigateToURL(
            new URLRequest(event.currentTarget.getRepeaterItem().url), '_blank');"/>
      </mx:Repeater>
  </mx:VBox>
</s:Application>
```

When executed, this example yields two buttons: one for Flex and one for Adobe® Flash®. When you click the button of your choice, the relevant product page loads in a new browser window.

Accessing specific instances of repeated components

Repeated components and repeated Repeater components have three properties that you can use to dynamically keep track of specific instances of repeated objects, to determine which Repeater component produced them, and to determine which `dataProvider` items were used by each Repeater component. The following table describes these properties:

Property	Description
instanceIndices	Array that contains the indices required to reference the component from its document. This Array is empty unless the component is in one or more Repeater components. The first element corresponds to the outermost Repeater component. For example, if the `id` is `b` and `instanceIndices` is `[2,4]`, you would reference it on the document as `b[2][4]`.
repeaters	Array that contains references to the Repeater components that produced the component. The Array is empty unless the component is in one or more Repeater components. The first element corresponds to the outermost Repeater component.
repeaterIndices	Array that contains the indices of the items in the `dataProvider` properties of the Repeater components that produced the component. The Array is empty unless the component is within one or more Repeater components. The first element corresponds to the outermost Repeater component. For example, if `repeaterIndices` is `[2,4]`, the outer Repeater component used its `dataProvider[2]` data item and the inner Repeater component used its `dataProvider[4]` data item. This property differs from `instanceIndices` if the `startingIndex` of any of the Repeater components is not 0. For example, even if a Repeater component starts at `dataProvider` item 4, the document reference of the first repeated component is `b[0]`, not `b[4]`.

The following example application uses the `repeaters` property to display the `id` value of the Repeater components in an Alert control when the user clicks one of the Button controls labelled by the index value of the outer Repeater component and inner Repeater component, respectively. It uses the `repeaters` property to get the `id` value of the inner Repeater component:

```
<?xml version="1.0"?>
<!-- repeater\RepeatersProp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      [Bindable]
      public var myArray:Array=[1,2];
    ]]>
  </fx:Script>
  <fx:Declarations>
    <s:ArrayCollection id="myAC" source="{myArray}"/>
  </fx:Declarations>

  <mx:VBox>
      <mx:Repeater  id="repeater1" dataProvider="{myAC}">
        <mx:Repeater  id="repeater2" dataProvider="{myAC}">
          <mx:Button
            label="[{repeater1.currentIndex},{repeater2.currentIndex}]"
            click="Alert.show(event.target.repeaters[1].id);"/>
        </mx:Repeater>
      </mx:Repeater>
  </mx:VBox>
</s:Application>
```

The following example application uses the `instanceIndices` property to set the `text` property of a TextInput control when the user clicks the corresponding Button control in the set of repeated Button and TextInput controls. You need to use the `instanceIndices` property because you must get the correct object dynamically; you cannot get it by its `id` value.

The following example shows how to use the `instanceIndices` property to set the `text` property of a TextInput control when the user clicks a Button control. The argument `event.target.instanceIndices` gets the index of the corresponding TextInput control.

```
<?xml version="1.0"?>
<!-- repeater\InstanceIndicesProp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
      [Bindable]
      public var myArray:Array=[1,2,3,4,5,6,7,8];
    ]]>
  </fx:Script>
  <fx:Declarations>
    <mx:ArrayCollection id="myAC" source="{myArray}"/>
  </fx:Declarations>

  <mx:Box>
      <mx:Repeater id="list" dataProvider="{myAC}" count="4" startingIndex="2">
        <mx:HBox>
          <mx:Button label="Click Me"
            click="myText[event.target.instanceIndices].text=
            event.target.instanceIndices.toString();"/>
          <mx:TextInput id="myText"/>
        </mx:HBox>
      </mx:Repeater>
  </mx:Box>
</s:Application>
```

The following code shows how to use the `repeaterIndices` property instead of the `instanceIndices` property to set the `text` property of a TextInput control when the user clicks a Button control. The value of `event.target.repeaterIndices` is based on the current index of the Repeater component. Because the `startingIndex` property of the Repeater component is set to 2, it does not match the `event.target.instanceIndices` value, which always starts at 0.

```
<?xml version="1.0"?>
<!-- repeater\RepeaterIndicesProp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
      [Bindable]
      public var myArray:Array = [1,2,3,4,5,6,7,8];
    ]]>
  </fx:Script>
  <mx:Box>
      <mx:Repeater id="list" dataProvider="{myArray}"
        startingIndex="2" count="4">
            <mx:HBox>
                <mx:Button id="b" label="Click Me"
                    click="myText[event.target.repeaterIndices-list.startingIndex].text=
                    event.target.repeaterIndices.toString();"/>
                <mx:TextInput id="myText"/>
            </mx:HBox>
      </mx:Repeater>
  </mx:Box>
</s:Application>
```

In this case, clicking the button prints the current element of the data provider, not the current iteration of the loop.

***Note:** The value of the `repeaterIndices` property is equal to the `startingIndex` property on the first iteration of the Repeater component. Subtracting the `startingIndex` value from the `repeaterIndices` value always yields the `instanceIndices` value.*

Using a Repeater component in a custom MXML component

You can use the `<mx:Repeater>` tag in an MXML component definition in the same way that you use it in an application file. When you use the MXML component as a tag in another MXML file, the repeated items appear. You can access an individual repeated item by its array index number, just as you do for a repeated item defined in the application file.

In the following example, a Button control in an MXML component called `childComp` is repeated for every element in an Array object called dp:

```
<?xml version="1.0"?>
<!-- repeater\myComponents\CustButton.mxml -->
<mx:VBox
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
      [Bindable]
      public var dp:Array=[1,2,3,4];
    ]]>
  </fx:Script>
  <fx:Declarations>
      <s:ArrayCollection id="myAC" source="{dp}"/>
  </fx:Declarations>

  <mx:Box>
      <mx:Repeater id="r" dataProvider="{myAC}">
        <mx:Button id="repbutton" label="button {r.currentItem}"/>
      </mx:Repeater>
  </mx:Box>
</mx:VBox>
```

The application file in the following example uses the childComp component to display four Button controls, one for each element in the array. The `getLabelRep()` function displays the label text of the second Button in the array.

```
<?xml version="1.0"?>
<!-- repeater\RepeatCustButton.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      public function getLabelRep():void {
        Alert.show(comp.repbutton[1].label);
      }
    ]]>
  </fx:Script>

  <MyComp:CustButton id="comp"/>
  <s:Button label="Get label of Repeated element" width="200"
    click="getLabelRep();"/>
</s:Application>
```

Dynamically creating components based on data type

You can use a Repeater component to dynamically create different types of components for specific items in a set of data. A Repeater component broadcasts a `repeat` event as it executes, and this event is broadcast after the `currentIndex` and `currentItem` properties are set. You can call an event handler function on the `repeat` event, and dynamically create different types of components based on the individual data items.

How a Repeater component executes

A Repeater component executes initially when it is instantiated. If the Repeater component's dataProvider property exists, it proceeds to instantiate its children, and they instantiate their children, recursively.

The Repeater component re-executes whenever its dataProvider, startingIndex, or count properties are set or modified either explicitly in ActionScript, or implicitly by data binding. If the dataProvider property is bound to a web service result, the Repeater component re-executes when the web service operation returns the result. A Repeater component also re-executes in response to paging through the dataProvider property by incrementally increasing the startingIndex value, as the following example shows:

```
r.startingIndex += r.count;
```

When a Repeater component re-executes, it destroys any children that it previously created (assuming the recycleChildren property is set to false), and then reinstantiates its children based on the current dataProvider property. The number of children in the container might change, and the container layout changes to accommodate any changes to the number of children.

Recreating children in a Repeater component

The recycleChildren property controls whether children of a Repeater component are recreated when the Repeater component re-executes. When you set the recycleChildren property to false, the Repeater component recreates all the objects when you swap one dataProvider with another, or sort, which causes a performance lag. Only set this property to true if you are confident that modifying your dataProvider will not recreate the Repeater component's children.

The default value of the recycleChildren property is false, to ensure that you do not leave stale state information in a repeated instance. For example, suppose you use a Repeater component to display photo images, and each Image control has an associated NumericStepper control for how many prints you want to order. Some of the state information, the image, comes from the dataProvider, while other state information, the print count, is set by user interaction. If you set the recycleChildren property to true and page through the photos by incrementally increasing the Repeater component's startingIndex value, the Image controls bind to the new images, but the NumericStepper controls keep the old information.

Considerations when using a Repeater component

Consider the following when you use a Repeater component:

- You must wrap a Repeater control in a MX container tag such as HBox, VBox, Box, Panel, or Canvas. The Repeater control cannot be a top-level child of the Application tag, and it cannot be a child of Spark containers such as Group, VGroup, or HGroup. If you are using nested repeaters, only the outer Repeater must be wrapped in a MX container.
- You cannot use a Repeater component to iterate through a two-dimensional Array object that is programmatically generated. This is because the elements of an Array object do not trigger changeEvent events, and therefore cannot function as binding sources at run time. Binding copies initial values during instantiation after variables are declared in an <mx:Script> tag, but before initialize handlers are executed.
- Run-time changes to an array used as a data provider are not reflected in the Repeater component. Use a collection if you need to allow run-time modification.
- Forgetting curly braces ({ }) in a dataProvider property is a common mistake when using a Repeater component. If the Repeater component doesn't execute, make sure that the binding is correct.
- If repeated objects are displayed out of order and you are using adjacent or nested Repeater components, you might need to place a dummy UIComponent immediately after the Repeater that is displaying objects incorrectly.

The code in the following example contains adjacent `<mx:Repeater>` tags and uses an `<mx:Spacer>` tag to create a dummy UIComponent:

```
<mx:VBox>
    <mx:Repeater id="r1">
        ...
    </mx:Repeater>
    <mx:Repeater id="r2">
        ...
    </mx:Repeater>
    <mx:Spacer height="0" id="dummy"/>
</mx:VBox>
```

The code in the following example contains nested `<mx:Repeater>` tags and uses an `<mx:Spacer>` tag to create a dummy UIComponent:

```
<mx:VBox>
    <mx:Repeater id="outer">
        <mx:Repeater id="inner">
            ...
        </mx:Repeater>
        <mx:Spacer id="dummy" height="0"/>
    </mx:Repeater>
</mx:VBox>
```

Chapter 5: Using data-driven UI components

Representing data

Data representation is a combination of features that provide a powerful way to validate, format, store, and pass data between objects.

About data representation

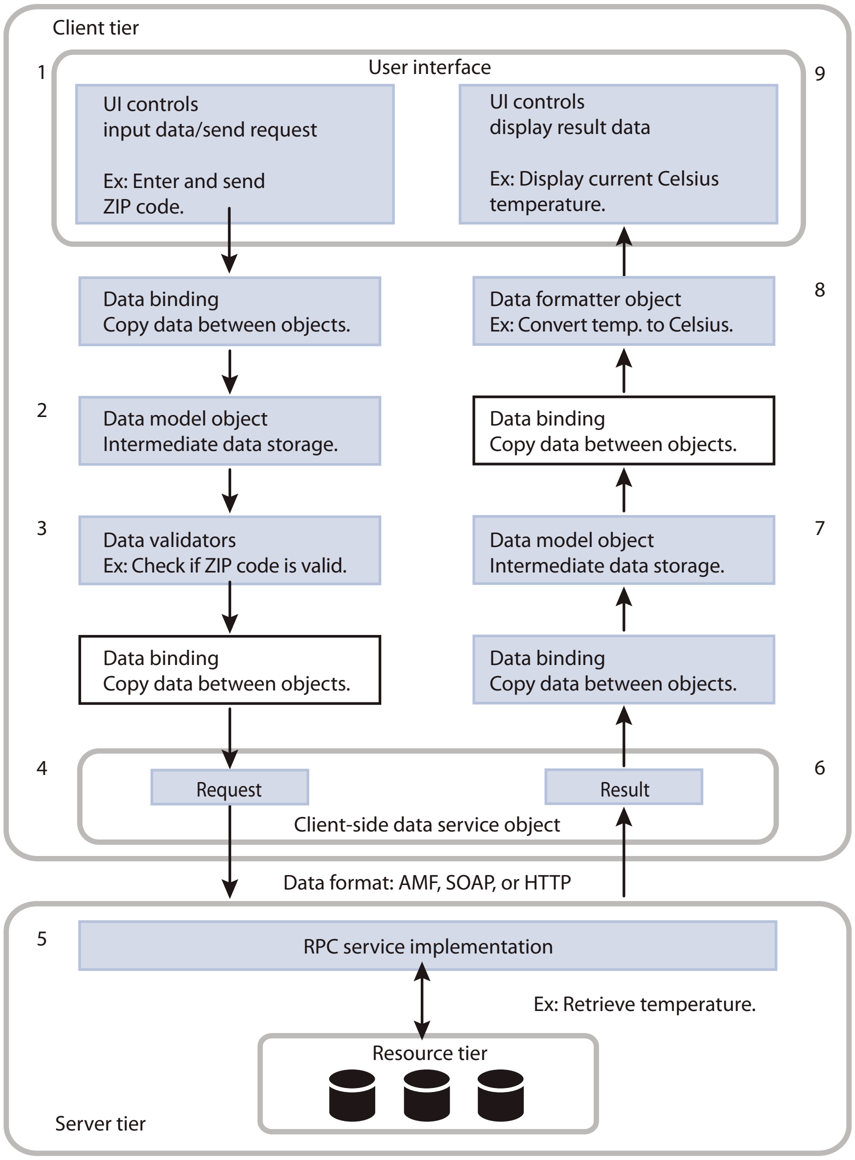
Adobe® Flex™ provides the following set of features for representing data in your applications: data binding, validation, and formatting. These features work in conjunction with the Adobe® LiveCycle™ Data Services ES features for working with remote data. Together, they allow you to perform the following tasks:

- Pass data between client-side objects.
- Store data in client-side objects.
- Validate data before passing it between client-side objects.
- Format data before displaying it.

The following steps describe a simple scenario in which a user provides input data and requests information in an Adobe Flex application:

1. The user enters data in input fields and submits a request by clicking a Button control.
2. (Optional) *Data binding* passes data to a data model object, which provides intermediate data storage. This allows data to be manipulated and passed to other objects in the application.
3. (Optional) One or more *data validator* objects validate the request data. Validator objects check whether the data meets specific criteria.
4. The data is passed to a server-side object.
5. The server-side object processes the request and returns data or a fault object if a valid result cannot be returned.
6. (Optional) Data binding passes data to a data model object, which provides intermediate data storage. This allows data to be manipulated and passed to other objects in the application.
7. (Optional) One or more *data formatter* objects format result data for display in the user interface.
8. Data binding passes data into user interface controls for display.

The following diagram illustrates what happens in Flex for two different user input examples. In one example, the user enters ZIP code data, and Flex validates the format. In the other example, the user requests the current temperature in Celsius.


Client tier
User interface
1
UI controls
input data/send request
Ex: Enter and send
ZIP code.
UI controls
display result data
Ex: Display current Celsius
temperature.
9
Data binding
Copy data between objects.
Data formatter object
Ex: Convert temp. to Celsius.
8
2
Data model object
Intermediate data storage.
Data binding
Copy data between objects.
3
Data validators
Ex: Check if ZIP code is valid.
Data model object
Intermediate data storage.
7
Data binding
Copy data between objects.
Data binding
Copy data between objects.
4
Request
Result
6
Client-side data service object
Data format: AMF, SOAP, or HTTP
5
RPC service implementation
Ex: Retrieve temperature.
Resource tier
Server tier

Data binding

The data binding feature provides a syntax for automatically copying the value of a property of one client-side object to a property of another object at run time. Data binding is usually triggered when the value of the source property changes. You can use data binding to pass user input data from user interface controls to a data service. You can also use data binding to pass results returned from a data service to user interface controls.

The following example shows a Text control that gets its data from an HSlider control's `value` property. The property name inside the curly braces ({ }) is a binding expression that copies the value of the source property, `mySlider.value`, into the Text control's `text` property.

```
<mx:HSlider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

For more information, see “Data binding” on page 299.

Data models

The data model feature lets you store data in client-side objects. A *data model* is an ActionScript object that contains properties for storing data, and that optionally contains methods for additional functionality. Data models are useful for partitioning the user interface and data in an application.

You can use the data binding feature to bind user interface data into a data model. You can also use the data binding feature to bind data from a data service to a data model.

You can define a simple data model in an MXML tag. When you require functionality beyond storage of untyped data, you can use an ActionScript class as a data model.

The following example shows an MXML-based data model with properties of TextInput controls bound into its fields:

```
<?xml version="1.0"?>
<!-- datarep\DatarepModelTag.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
         <fx:Model id="reg">
           <registration>
              <name>{nme.text}</name>
              <email>{email.text}</email>
              <phone>{phone.text}</phone>
              <zip>{zip.text}</zip>
              <ssn>{ssn.text}</ssn>
          </registration>
        </fx:Model>
    </fx:Declarations>

    <s:TextInput id="nme"/>
    <s:TextInput id="email"/>
    <s:TextInput id="phone"/>
    <s:TextInput id="zip"/>
    <s:TextInput id="ssn"/>
</s:Application>
```

For more information about data models, see “Storing data” on page 889.

Data validation

The data validation feature lets you ensure that data meets specific criteria before the application uses the data. *Data validators* are ActionScript objects that check whether data in a component is formatted correctly. You can apply a data validator to a property of any component. For models in a remote procedure call (RPC) component declaration, properties to which a validator component is applied are validated just before the request is sent to an RPC service destination. Only valid requests are sent.

The following example shows MXML code that uses the standard ZipCodeValidator component, represented by the `<mx:ZipCodeValidator>` tag, to validate the format of the ZIP code that a user enters. The `source` property of the ZipCodeValidator validator indicates the property that it validates.

```
<?xml version="1.0"?>
<!-- datarep\Validate.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:Model id="zipModel">
            <root>
                <zip>{input.text}</zip>
            </root>
        </fx:Model>
        <mx:ZipCodeValidator source="{zipModel}"
            property="zip"
            listener="{input}"
            trigger="{input}"/>
    </fx:Declarations>
    <s:TextInput id="input"
        text="enter zip"
        width="80"/>
    <s:TextInput
        text="click here to validate"/>
</s:Application>
```

For more information about validator components, see “Validating Data” on page 1964.

Data formatting

The data formatting feature lets you change the format of data before displaying it in a user interface control. For example, when a data service returns a string that you want to display in the (xxx)xxx-xxxx phone number format, you can use a formatter component to ensure that the string is reformatted before it is displayed.

A *data formatter component* is an object that formats raw data into a customized string. You can use data formatter components with data binding to reformat data that is returned from a data service.

The following example formats the value of the `text` property of a TextInput control into a long date format:

```
<?xml version="1.0"?>
<!-- datarep\Format.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Declare a DateTimeFormatter and define the dateStyle.-->
        <s:DateTimeFormatter id="dateTimeFormatter" dateStyle="long"/>
    </fx:Declarations>
    <s:Label text="Enter date (mm/dd/yyyy):"/>
    <s:TextInput id="dob" text="12/08/1980" width="300"/>
    <s:Label text="Formatted date: "/>
    <s:TextInput id="formattedDate"
        text="" editable="false" width="300"/>

    <!-- Format and update the date.-->
    <s:Button label="Format Input"
        click="formattedDate.text=dateTimeFormatter.format(dob.text);"/>
</s:Application>
```

For more information about data formatters, see “Formatting Data” on page 2004.

Storing data

You use the data model feature to store data in an application before it is sent to the server, or to store data sent from the server before using it in the application.

About data models

A *data model* is an ActionScript object that contains properties that you use to store application-specific data. Communication between an Adobe® Flex™ application and the server is required only to retrieve data not yet available to the Flex application and to update a server-side data source with new data from the Flex application.

You can use a data model for data validation, and it can contain client-side business logic. You can define a data model in MXML or ActionScript. In the model-view-controller (MVC) design pattern, the data model represents the model tier.

When you plan an application, you determine the kinds of data that the application must store and how that data must be manipulated. This helps you decide what types of data models you need. For example, suppose you decide that your application must store data about employees. A simple employee model might contain name, department, and e-mail address properties.

Defining a data model

You can define a data model in an MXML tag, an ActionScript function, or an ActionScript class. In general, you should use MXML-based models for simple data structures, and use ActionScript for more complex structures and client-side business logic.

***Note:** The `<fx:Model>` and `<fx:XML>` tags are Flex compiler tags and do not correspond directly to ActionScript classes. The ActionScript 3.0 Reference for the Adobe Flash Platform contains information about these tags and other compiler tags.*

You can place an `<fx:Model>` tag or an `<fx:XML>` tag in a Flex application file or in an MXML component file. The tag should have an `id` value, and it cannot be the root tag of an MXML component.

Declare an `<fx:Model>` tag or an `<fx:XML>` tag in an `<fx:Declarations>` tag. You define these tags in an `<fx:Declarations>` tag because they are not visual components.

The <fx:Model> tag

The most common type of MXML-based model is the <fx:Model> tag, which is compiled into an ActionScript object of type mx.utils.ObjectProxy, which contains a tree of objects when your data is in a hierarchy, with no type information. The leaves of the Object tree are scalar values. Because models that are defined in `<fx:Model>` tags contain no type information or business logic, you should use them only for the simplest cases. Define models in ActionScript classes when you need the typed properties or you want to add business logic.

The following example shows an employee model declared in an `<fx:Model>` tag:

```
<?xml version="1.0"?>
<!-- Models\ModelsModelTag.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <fx:Model id="employeemodel">
            <employee>
                <name>
                    <first/>
                    <last/>
                </name>
                <department/>
                <email/>
            </employee>
        </fx:Model>
    </fx:Declarations>
</s:Application>
```

An `<fx:Model>` child tag with no value is considered null. If you want an empty string instead, you can use a binding expression as the value of the tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- Models\ModelTagEmptyString.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <fx:Model id="employeemodel">
            <employee>
                <name>
                    <!--Fill the first property with empty string.-->
                    <first>{""}</first>
                    <!--Fill the last property with empty string.-->
                    <last>{""}</last>
                </name>
                <!--department is null-->
                <department/>
                <!--email is null-->
                <email/>
            </employee>
        </fx:Model>
    </fx:Declarations>
</s:Application>
```

The <fx:XML> tag

An <fx:XML> tag represents literal XML data. Setting the `format` property to `e4x` creates an XML object, which implements the powerful XML-handling standards defined in the ECMAScript for XML specification (ECMA-357 edition 2) (known as E4X). For backward compatibility, when the `format` property is not explicitly set to `e4x`, the type of the object created is flash.xml.XMLNode.

***Note:** You currently cannot use a node within an `<fx:XML>` data model as a binding source.*

Script-based models

As an alternative to using an MXML-based model, you can define a model as a variable in an <fx:Script> tag. The following example shows a very simple model defined in an ActionScript script block. It would be easier to declare this model in an `<fx:Model>` tag.

```
<?xml version="1.0"?>
<!-- Models\ScriptModel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            [Bindable]
            public var myEmployee:Object={
                name:{
                    first:null,
                    last:null
                    },
                department:null,
                email:null
            };
        ]]>
    </fx:Script>
</s:Application>
```

There is no advantage to using a script-based model instead of an MXML-based model. As with MXML-based models, you cannot type the properties of a script-based model. To type properties, you must use a class-based model.

Class-based models

Using an ActionScript class as a model is a good option when you want to store complex data structures with typed properties, or when you want to execute client-side business logic by using application data. Also, the type information in a class-based model is retained on the server when the model is passed to a server-side data service.

The following example shows a model defined in an ActionScript class. This model is used to store shopping cart items in an e-commerce application. It also provides business logic in the form of methods for adding and removing items, getting an item count, and getting the total price.

```
package
{

[Bindable]
public class ShoppingCart {
    public var items:Array = [];

    public var total:Number = 0;

    public var shippingCost:Number = 0;

    public function ShoppingCart() {
    }

    public function addItem(item:Object, qty:int = 1,
        index:int = 0):void {
        items.splice(index, 0, { id: item.id,
        name: item.name,
        description: item.description,
        image: item.image,
        price: item.price,
        qty: qty });
        total += parseFloat(item.price) * qty;
}

    public function removeItemAt(index:Number):void {
        total -= parseFloat(items[index].price) * items[index].qty;
        items.splice(index, 1);
        if (getItemCount() == 0)
        shippingCost = 0;
}

    public function getItemCount():int {
        return items.length;
    }

    public function getTotal():Number {
        return total;
    }
}
}
```

Note: *You can use properties of any recognized type in a class-based model. For example, you could create an Employee class, and then define properties of type Employee in another class that imports Employee.*

You declare a class-based model as an ActionScript component tag in an MXML file, as the following example shows:

```
<local:ShoppingCart id="cart" xmlns:local="*"/>
```

This component is in the same directory as the MXML file, as indicated by the XML namespace value `*`. For more information about specifying the location of components, see “Using XML namespaces” on page 10.

Specifying an external source for an <fx:Model> tag or <fx:XML> tag

You can specify an external source for an <fx:Model> or <fx:XML> tag in a `source` property. Separating the content of a model from the MXML that defines the user interface improves the maintainability and reusability of an application. Adobe recommends this way of adding static XML content to a Flex application.

The external source file can contain static data and data binding expressions, just like a model defined in the body of the `<fx:Model>` or `<fx:XML>` tag. The file referenced in a `source` property resides on the server and not on the client machine. The compiler reads the `source` value and compiles the source into the application; the `source` value is not read at run time. To retrieve XML data at run time, you can use the `<mx:HTTPService>` tag; for more information, see Accessing Server-Side Data with Flex.

Using `<fx:Model>` and `<fx:XML>` tags with external sources is an easy way to reuse data model structures and data binding expressions. You can also use them to prepopulate user interface controls with static data by binding data from the model elements into the user interface controls.

The `source` property accepts the names of files relative to the current web application directory, as well as URLs with HTTP:// prefixes. In the following example, the content of the myEmployee1 data model is an XML file named content.xml in the local web application directory. The content of the myEmployee2 data model is a fictional HTTP URL that returns XML.

```
<fx:Model source="employees.xml" id="employee1"/>

<fx:Model source="http://www.somesite1.com/employees.xml" id="employee2"/>
```

The source file must be a valid XML document with a single root node. The following example shows an XML file that could be used as the source of the `<fx:Model source="employees.xml" id="Model1"/>` tag.

```
<?xml version="1.0"?>
<employees>
    <employee>
        <name>John Doe</name>
        <phone>555-777-66555</phone>
        <email>jdoe@fictitious.com</email>
        <active>true</active>
    </employee>
    <employee>
        <name>Jane Doe</name>
        <phone>555-777-66555</phone>
        <email>jndoe@fictitious.com</email>
        <active>true</active>
    </employee>
</employees>
```

Using validators with a data model

To validate the data stored in a data model, you use validators. In the following example, the `<mx:EmailValidator>`, `<mx:PhoneNumberValidator>`, `<mx:ZipCodeValidator>`, and `<mx:SocialSecurityValidator>` tags declare validators that validate the e-mail, phone, zip, and ssn fields of the registration data model. The validators generate error messages when a user enters incorrect data in TextInput controls that are bound to the data model fields.

```
<?xml version="1.0"?>
<!-- Models\ModelWithValidator.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <fx:Model id="reg">
            <registration>
                <name>{username.text}</name>
                <email>{email.text}</email>
                <phone>{phone.text}</phone>
                <zip>{zip.text}</zip>
                <ssn>{ssn.text}</ssn>
            </registration>
        </fx:Model>

        <mx:Validator required="true"
            source="{reg}" property="name"
            trigger="{submit}"
            triggerEvent="click"
            listener="{username}"/>
        <mx:EmailValidator source="{reg}" property="email"
            trigger="{submit}"
            triggerEvent="click"
            listener="{email}"/>
        <mx:PhoneNumberValidator source="{reg}" property="phone"
            trigger="{submit}"
            triggerEvent="click"
            listener="{phone}"/>
        <mx:ZipCodeValidator source="{reg}" property="zip"
            trigger="{submit}"
            triggerEvent="click"
            listener="{zip}"/>
        <mx:SocialSecurityValidator source="{reg}" property="ssn"
            trigger="{submit}"
            triggerEvent="click"
            listener="{ssn}"/>
    </fx:Declarations>

    <!-- Form contains user input controls. -->
    <s:Form>
        <s:FormItem label="Name" required="true">
            <s:TextInput id="username" width="200"/>
        </s:FormItem>
        <s:FormItem label="Email" required="true">
            <s:TextInput id="email" width="200"/>
        </s:FormItem>
        <s:FormItem label="Phone" required="true">
            <s:TextInput id="phone" width="200"/>
        </s:FormItem>
        <s:FormItem label="Zip" required="true">
            <s:TextInput id="zip" width="60"/>
        </s:FormItem>
        <s:FormItem label="Social Security" required="true">
            <s:TextInput id="ssn" width="200"/>
        </s:FormItem>
        <s:FormItem>
        <!-- User clicks Button to trigger validation. -->
            <s:Button id="submit" label="Validate"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

This example cleanly separates the user interface and application-specific data. You could easily extend it to create a three-tier architecture by binding data from the registration data model into an RPC service request. You could also bind user input data directly into an RPC service request, which itself is a data model, as described in Accessing Server-Side Data with Flex.

For more information about validators, see “Validating Data” on page 1964.

Using a data model as a value object

You can use a data model as a value object, which acts as a central repository for a set of data returned from method calls on one or more objects. This makes it easier to manage and work with data in an application.

In the following example, the tentModel data model stores the results of a web service operation. The TentDetail component is a custom MXML component that gets its data from the tentModel data model and displays details for the currently selected tent.

```
...
<!-- Data model stores data from selected tent. -->
<fx:Model id="tentModel">
    <tent>
        <name>{selectedTent.name}</name>
        <sku>{selectedTent.sku}</sku>
        <capacity>{selectedTent.capacity}</capacity>
        <season>{selectedTent.seasonStr}</season>
        <type>{selectedTent.typeStr}</type>
        <floorarea>{selectedTent.floorArea}</floorarea>
        <waterproof>{getWaterProof(selectedTent.waterProof)}</waterproof>
        <weight>{getWeight(selectedTent)}</weight>
        <price>{selectedTent.price}</price>
    </tent>
</fx:Model>
...
    <TentDetail id="detail" tent="{tentModel}"/>
...
```

The following example shows the MXML source code for the TentDetail component. References to the `tent` property, which contains the tentModel data model, and the corresponding `tentModel` properties are highlighted in boldface.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Models\myComponents\TentDetail.mxml-->
<s:Panel xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    title="Tent Details">

    <fx:Script>
        <![CDATA[
            [Bindable]
            public var tent:Object;
        ]]>
    </fx:Script>

    <fx:Style>
        .title{fontFamily:Arial;fontWeight:bold;color:#3D3D3D;fontSize:16pt;}
        .flabelColor
            {fontFamily:Arial;fontWeight:bold;color:#3D3D3D;fontSize:11pt}
        .productSpec{fontFamily:Arial;color:#5B5B5B;fontSize:10pt}
    </fx:Style>

    <s:VGroup paddingLeft="10" paddingTop="10" paddingRight="10">
        <s:Form verticalGap="0" paddingLeft="10" paddingTop="10"
            paddingRight="10" paddingBottom="0">

            <s:VGroup width="209" height="213">
                <s:Image width="207" height="211"
                    source="./images/{tent.sku}_detail.jpg"/>
            </s:VGroup>

            <s:FormHeading label="{tent.name}" paddingTop="1"
                styleName="title"/>

            <mx:HRule width="209"/>
            <s:FormItem label="Capacity" styleName="flabelColor">
                <s:Label text="{tent.capacity} person"
                    styleName="productSpec"/>
            </s:FormItem>
            <s:FormItem label="Season"
                styleName="flabelColor">
                <s:Label text="{tent.season}"
                    styleName="productSpec"/>
            </s:FormItem>
            <s:FormItem label="Type" styleName="flabelColor">
                <s:Label text="{tent.type}"
                    styleName="productSpec"/>
            </s:FormItem>
            <s:FormItem label="Floor Area" styleName="flabelColor">
                <s:Label text="{tent.floorarea}
                    square feet" styleName="productSpec"/>
            </s:FormItem>
            <s:FormItem label="Weather" styleName="flabelColor">
                <s:Label text="{tent.waterproof}"
                    styleName="productSpec"/>
            </s:FormItem>
            <s:FormItem label="Weight" styleName="flabelColor">
                <s:Label text="{tent.weight}"
                    styleName="productSpec"/>
            </s:FormItem>
        </s:Form>
    </s:VGroup>
</s:Panel>
```

Binding data into an XML data model

Flex compiles the `<fx:XML>` tag into literal XML data in an ActionScript xml.XMLNode or XML object. This is different from the `<fx:Model>` tag, which Flex compiles into an Action object that contains a tree of ActionScript objects. To bind data into an `<fx:XML>` data model, you can use the curly braces syntax the same way you do with other data models. However, you cannot use a node within the data model as a binding source.

Adobe does not recommend using the `<mx:Binding>` tag for this type of binding because doing so requires you to write an appropriate ActionScript XML command as the `destination` property of the `<mx:Binding>` tag. For more information about the `<fx:XML>` tag, see “Defining a data model” on page 889.

The following example shows an `<fx:XML>` data model with binding destinations in curly braces:

```
<?xml version="1.0"?>
<!-- Models\XMLBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <fx:XML id="myEmployee" format="e4x">
            <employee>
                <name>
                    <first>{firstName.text}</first>
                    <last>{lastName.text}</last>
                    </name>
                <department>{department.text}</department>
                <email>{email.text}</email>
            </employee>
        </fx:XML>
    </fx:Declarations>
    <s:TextInput id="firstName"/>
    <s:TextInput id="lastName"/>
    <s:TextInput id="department"/>
    <s:TextInput id="email"/>
</s:Application>
```

Data providers and collections

A *collection* object contains a data object, such as an Array or an XMList object, and provides a set of methods that let you access, sort, filter, and modify the data items in that data object. Several controls, known as *data provider controls*, have a `dataProvider` property that you populate with a collection.

The mx.collections.ArrayCollection and mx.collections.XMLListCollection classes are specific collection implementations that you can use with any data provider control in the Flex framework. Most data provider controls also support using the mx.collections.ArrayList class. This class is similar to the ArrayCollection class, except that is uses less memory, but does not have as many features.

For more information on Flex components that use data providers, see “Spark list-based controls” on page 514, “MX data-driven controls” on page 943, and “Menu-based controls” on page 985.

About collections and data provider components

Data provider components require data for display or user interaction. To provide this data, you assign a collection, which is usually an ArrayCollection, ArrayList, or XMLListCollection object, to the data provider component’s `dataProvider` property.

Optionally, for MX controls only, you can assign a raw data object such as an Array, XML, or XMLList object to the data provider component’s `dataProvider` property; however, this is not considered a best practice because of the limitations noted in “About data provider components” on page 901.

The source of data in a collection object can be local or a remote source, such as a web service or a PHP page that you call with a Flex data access component.

About collections

Collections are objects that provide a uniform way to access and represent the data contained in a data source object, such as an Array or an XMLList object. Collections provide a level of abstraction between components and the data objects that you use to populate them.

The standard collection types in the Flex framework, the ArrayCollection and XMLListCollection classes, extend the mx.collections.ListCollectionView class, which implements the mx.collections.ICollectionView and mx.collections.IList interfaces. These interfaces provide the underlying functionality for viewing and modifying data objects. An ArrayCollection object takes an Array as its source object. An XMLListCollection object take an XMLList object as its source object.

Collections provide the following features:

- Ensure that a component is properly updated when the underlying data changes. Components are not updated when noncollection data objects change. (They *are* updated to reflect the new data the next time they are refreshed.) If the data provider is a collection, the components are updated immediately after the collection change occurs.
- Provide mechanisms for handling paged data from remote data sources that may not initially be available and may arrive over a period of time.
- Provide a consistent set of operations on the data, independent of those provided by the raw data source. For example, you can insert and delete objects by using an index into the collection, independently of whether the underlying data is, for example, in an Array or an Object.
- Provide a specific *view* of the data that can be in sorted order, or filtered by a developer-supplied method. This is only a view of the data; it does not change the data.
- Use a single collection to populate multiple components from the same data source.
- Use collections to switch data sources for a component at run time, and to modify the content of the data source so that changes are reflected by all components that use the data source.
- Use collection methods to access data in the underlying data source.

***Note:** If you use a raw data object, such as an Array, as the value of an MX control's data provider, Flex automatically wraps the object in a collection wrapper (either an ArrayCollection or XMLListCollection). The control does not automatically detect changes that are made directly to the raw object. A change in the length of an array, for example, does not result in an update of the control. You can, however, use an object proxy, a listener interface, or the* `itemUpdated()` *method to notify the view of certain changes. For Spark controls, you cannot use a raw object as the value of the control's data provider. You must specify an object that implements the IList interface. Classes that implement IList include ArrayCollection, ArrayList, and XMLListCollection.*

Another type of collection, ArrayList, extends the IList interface but not the ICollectionView interface. As a result, it is more lightweight and provides most of the same functionality as the ArrayCollection class. It does not, however, support sorting, filtering, or cursors.

Collection interfaces

Collections use the following interfaces to define how a collection represents data and provides access to it. The standard Flex framework collections, the ArrayCollection and XMLListCollection classes, implement both the ICollectionView interface and the IList interface. The IList and ICollectionView interfaces provide alternate methods for accessing and changing data. The IList interface is simpler; it provides add, set, get, and remove operations that operate directly on linear data.

Interface	Description
IList	A direct representation of items organized in an ordinal fashion. The interface presents the data from the source object in the same order as it exists in that object, and provides access and manipulation methods based on an index. The IList class does not provide sorting, filtering, or cursor functionality.
ICollectionView	A view of a collection of items. You can modify the view to show the data in sorted order and to show a subset of the items in the source object, as specified by a filter function. A class that implements this interface can use an IList interface as the underlying collection. The interface provides access to an IViewCursor object for access to the items.
IViewCursor	Enumerates an object that implements the ICollectionView interface bidirectionally. The *view cursor* provides find, seek, and bookmarking capabilities, and lets you modify the underlying data (and the view) by inserting and removing items.

The ICollectionView interface (also called the *collection view*) provides a more complex set of operations than the IList interface, and is appropriate when the underlying data is not organized linearly. Its data access techniques, however, are more complex than those of the IList interface, so you should use the IList interface if you need only simple, indexed access to a linear collection. The collection view can represent a subset of the underlying data, as determined by a sort or filter operation.

Collection classes

The following table describes the public classes in the mx.collections.* and spark.collections.* packages. It does not include constant, event, and error classes. For complete reference information on collection-related classes, see the MX collections, Spark collections, and collections.errors packages, and the CollectionEvent and CollectionEventKind classes in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Class	Description
ArrayCollection	A standard collection for working with Arrays. Implements the IList and ICollectionView interfaces.
ArrayList	A standard collection for working with Arrays. Implements the IList interface. You can use this class instead of the ArrayCollection class if you do not need to sort, filter, or use cursors in your collection.
AsyncListView	A standard collections that handles `ItemPendingErrors` thrown by the `getItemAt()`, `removeItemAt()`, and `toArray()` methods.Use this class to support data paging when accessing data from a remote server.
XMLListCollection	A standard collection for working with XMLList objects. Implements the IList and ICollectionView interfaces, and a subset of XMLList methods.
CursorBookmark	Represents the position of a view cursor within a collection. You can save a view cursor position in a CursorBookmark object and use the object to return the view cursor to the position at a later time.
Sort	Provides the information and methods required to sort a collection.
SortField	Provides properties and methods that determine how a specific field affects data sorting in a collection.
ItemResponder	(Used only if the data source is remote.) Handles cases when requested data is not yet available.
ListCollectionView	Superclass of the ArrayCollection and XMLListCollection classes. Adapts an object that implements the IList interface to the ICollectionView interface so that it can be passed to anything that expects an IList or an ICollectionView.

About data provider components

Several Flex components, including all list-based controls, are called data provider components because they have a `dataProvider` property that consumes data from an ArrayCollection, ArrayList, XMLListCollection object, or a custom collection. For example, the value of an MX Tree control's `dataProvider` property determines the structure of the tree and any associated data assigned to each tree node, and a ComboBox control's `dataProvider` property determines the items in the control's drop-down list. Many standard controls, including the ColorPicker and MenuBar controls, also have a `dataProvider` property.

For Spark list-based controls, the value of the `dataProvider` property must implement the IList interface. Classes that implement IList include ArrayCollection, ArrayList, and XMLListCollection.

For the MX list-based controls, you can specify raw data objects, such as an Array of strings or objects or an XML object, the value of the `dataProvider` property. For MX controls, Flex automatically wraps the raw data in a collection.

Adobe recommends that you always specify a collection as the value of the `dataProvider` property. Using collections explicitly ensures data synchronization and provides both simpler and more sophisticated data access and manipulation tools than are available when you are using raw objects directly as data providers. Collections can also provide a consistent interface for accessing and managing data of different types. For more information about collections, see "About collections and data provider components" on page 898.

Although raw data objects for MX controls are automatically wrapped in an ArrayCollection object or XMLListCollection, they are subject to the following limitations:

- Raw objects are often not sufficient if you have data that changes, because the data provider component does not receive a notification of any changes to the base object. The component therefore does not get updated until it must be redrawn due to other changes in the application, or if the data provider is reassigned. At that time, it gets the data again from the updated raw object.
- Raw objects do not provide advanced tools for accessing, sorting, or filtering data. For example, if you use an Array as the data provider, you must use the native Adobe® Flash® Array methods to manipulate the data.

For detailed descriptions of the individual controls, see the pages for the controls in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For information on programming with many of the data provider components, see "MX data-driven controls" on page 943.

Data objects

The Flex framework supports the following types of data objects for populating data provider components:

Linear or list-based data objects are flat data structures consisting of some number of objects, each of which has the same structure; they are often one-dimensional Arrays or ActionScript object graphs, or simple XML structures. You can specify one of these data structures in an ArrayList, ArrayCollection or XMLListCollection object's `source` property or as the value of a data provider control's `dataProvider` property.

You can use list-based data objects with all data provider controls, but you do not typically use them with Tree and most menu-based controls, which typically use hierarchical data structures. For data that can change dynamically, you typically use an ArrayCollection, ArrayList, or XMLListCollection object to represent and manipulate these data objects rather than the raw data object. You can also use a custom object that implements the ICollectionView and/or IList interfaces. If you do not require sorting, cursors, and filtering, you can use a class that implements just the IList interface

Hierarchical data objects consist of cascading levels of often asymmetrical data. Often, the source of the data is an XML object, but the source can be a generic Object tree or trees of typed objects. You typically use hierarchical data objects with Flex controls that are designed to display hierarchical data:

- Tree
- Menu
- MenuBar
- PopUpMenuButton

A hierarchical data object matches the layout of a tree or cascading menu. For example, a tree often has a root node, with one or more branch or leaf child nodes. Each branch node can hold additional child branch or leaf nodes, but a leaf node is an endpoint of the tree.

The Flex hierarchical data provider controls use *data descriptor* interfaces to access and manipulate hierarchical data objects, and the Flex framework provides one class, the DefaultDataDescriptor class, which implements the required interfaces. If your data object does not conform to the structural requirements of the default data descriptor, you can create your own data descriptor class that implements the required interfaces.

You can use an ArrayCollection object or XMLListCollection object, or a custom object that implements the ICollectionView and IList interfaces to access and manipulate dynamic hierarchical data.

You can also use a hierarchical data object with controls that take linear data, such as the List control and the DataGrid control, by extracting specific data for linear display.

For more information on using hierarchical data providers, see “Hierarchical data objects” on page 927.

Specifying data providers in MXML applications

The Flex framework lets you specify and access data for data provider components in many ways. For example, you can bind a collection that contains data from a remote source to the `dataProvider` property of a data provider component; you can define the data provider in a `dataProvider` child tag of a data provider component; or you can define the data provider in ActionScript.

All access techniques belong to one of the following patterns, whether data is local or is provided from a remote source:

- Using a collection implementation, such as an ArrayList object, ArrayCollection object, or XMLListCollection object, directly. This pattern is particularly useful for collections where object reusability is not important.
- Using a collection interface. This pattern provides the maximum of independence from the underlying collection implementation.
- Using a raw data object, such as an Array, with an MX list-based control. This pattern is discouraged unless data is completely static.

Using a collection object explicitly

You can use a collection, such as an ArrayList, ArrayCollection, or XMLListCollection object, as a data provider explicitly in an MXML control by assigning it to the component’s `dataProvider` property. This technique is more direct than using an interface and is appropriate if the data provider is always of the same collection type.

For list-based controls, you often use an ArrayList object as the data provider, and populate the ArrayList object by using an Array that is local or from a remote data source. The following example shows an ArrayList object declared in line in a ComboBox control:

```
<?xml version="1.0"?>
<!-- dpcontrols\ArrayCollectionInComboBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:ComboBox id="myCB">
        <s:ArrayList id="stateArray">
            <fx:Object label="AL" data="Montgomery"/>
            <fx:Object label="AK" data="Juneau"/>
            <fx:Object label="AR" data="Little Rock"/>
        </s:ArrayList>
    </s:ComboBox>
</s:Application>
```

In this example, the default property of the ComboBox control, `dataProvider`, defines an ArrayList object. Because `dataProvider` is the default property of the ComboBox control, it is not declared. The default property of the ArrayList object, `source`, is an Array of Objects, each of which has a label and a data field. Because the ArrayList object's `source` property takes an Array object, it is not necessary to declare the `<fx:Array>` tag as the parent of the `<fx:Object>` tags.

The following example uses ActionScript to declare and create an ArrayList object:

```
<?xml version="1.0"?>
<!-- dpcontrols\ArrayCollectionInAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">

    <fx:Script>
        <![CDATA[
            import mx.collections.*;
            [Bindable]
            public var stateArray:ArrayList;

            public function initData():void {
                stateArray=new ArrayList(
                [{label:"AL", data:"Montgomery"},
                {label:"AK", data:"Juneau"},
                {label:"AR", data:"Little Rock"}]);
            }
        ]]>
    </fx:Script>
    <s:ComboBox id="myComboBox" dataProvider="{stateArray}"/>
</s:Application>
```

After you define the ComboBox control, the `dataProvider` property of the ComboBox control provides access to the collection that represents the underlying source object, and you can use the property to modify the data provider. If you add the following button to the preceding code, for example, you can click the button to add the label and data for Arizona to the end of the list in the ComboBox control, as in the following example:

```
<s:Button label="Add AZ"
    click="stateArray.addItem({'label':'AZ', 'data':'Phoenix'});"/>
```

In many cases, an ArrayList class is adequate for defining a non-XML data provider. However, for web services, remote objects, and uses that require cursors, filters, and sorts, you use the ArrayCollection class.

The following example shows an ArrayCollection object that is populated from a remote data source, in this case a remote object, as the data provider of a DataGrid control. Note that the `ArrayUtil.toArray()` method is used to ensure that the data sent to the ArrayCollection object is an Array.

```
<?xml version="1.0"?>
<!-- dpcontrols\ROParamBind22.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout gap="10"/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.utils.ArrayUtil;
        ]]>
    </fx:Script>

    <fx:Declarations>
        <mx:RemoteObject
            id="employeeRO"
            destination="roDest"
            showBusyCursor="true"
            fault="Alert.show(event.fault.faultString, 'Error');">
            <mx:method name="getList">
                <mx:arguments>
                    <deptId>{dept.selectedItem.data}</deptId>
                </mx:arguments>
            </mx:method>
        </mx:RemoteObject>
        <mx:ArrayCollection id="employeeAC"
            source="{ArrayUtil.toArray(employeeRO.getList.lastResult)}"/>
    </fx:Declarations>
    <s:HGroup>
        <s:Label text="Select a department:"/>
        <s:ComboBox id="dept" width="150">
            <s:dataProvider>
                <mx:ArrayCollection>
```

```
                    <mx:source>
                        <fx:Object label="Engineering" data="ENG"/>
                        <fx:Object label="Product Management" data="PM"/>
                        <fx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </s:dataProvider>
        </s:ComboBox>
        <s:Button label="Get Employee List"
            click="employeeRO.getList.send()"/>
    </s:HGroup>

    <mx:DataGrid dataProvider="{employeeAC}" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Accessing data by using collection interfaces

If you know that a control's data can always be represented by a specific collection class, use an ArrayList, ArrayCollection, or XMLListCollection object explicitly, as shown above. If your code might be used with different types of collections—for example, if you might switch between object-based data and XML data from a remote data service—then you should use the ICollectionView interface in your application code, as the following example shows:

```
public var myICV:ICollectionView = indeterminateCollection;
...
<s:ComboBox id="cb1" dataProvider="{myICV}" initialize="sortICV()"/>
```

You can then manipulate the interface as needed to select data for viewing, or to get and modify the data in the underlying data object.

Using a raw data object as a data provider for an MX control

When the data is static, you can use a raw data object, such as an Array object, directly as a data provider for an MX control. For example, you could use an array for a static list of U.S. Postal Service state designators. Do not use the data object directly as the `dataProvider` property of a control if the object contents can change dynamically, for example in response to user input or programmatic processing.

Note: *For Spark list-based controls, you cannot use a raw object as the value of the control's data provider. You must specify an object that implements the IList interface. Classes that implement IList include ArrayCollection, ArrayList, and XMLListCollection.*

The result returned by an HTTP service or web service is often an Array, and if you treat that data as read-only, you can use the Array directly as a data provider. However, it is a better practice to use a collection explicitly. List-based controls turn Array-based data providers into collections internally, so there is no performance advantage to using an Array directly as the data provider. If you pass an Array to multiple controls, it is more efficient to convert the Array into a collection when the data is received, and then pass the collection to the controls.

The following example shows an MX ComboBox control that takes a static Array as its `dataProvider` value. As noted previously, using raw objects as data providers is not a best practice and should be considered only for data objects that will not change.

```
<?xml version="1.0"?>
<!-- dpcontrols/StaticComboBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            [Bindable]
            public var myArray:Array = ["AL", "AK", "AR"];
        ]]>
    </fx:Script>

    <mx:ComboBox id="myCB0" dataProvider="{myArray}"/>
</s:Application>
```

In the preceding example, the Array specified as the `dataProvider` is automatically wrapped in an ArrayCollection object. This is the default type for data providers to be wrapped in. If the data provider were an XML or XMLList object, it would be wrapped in an XMLListCollection object. In this example, the ArrayCollection does not have an `id` property, but you can use ArrayCollection methods and properties directly through the `dataProvider` property, as the following example shows:

```
<mx:Button label="Add AZ"
    click="myCB0.dataProvider.addItem({'label':'AZ','data':'Phoenix'});"/>
```

Setting a data provider in ActionScript

You may set the `dataProvider` property of a control in ActionScript, as well as in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridValidateNow.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
   initialize="initData();">
   <fx:Script>
       <![CDATA[

            import mx.collections.ArrayList;
            [Bindable]
            private var DGArray:ArrayList = new ArrayList([
                {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}]);

            // Initialize initDG ArrayList variable from the ArrayList.
            public function initData():void {
                myGrid.dataProvider = DGArray;
            }
        ]]>
   </fx:Script>
   <mx:DataGrid id="myGrid">
      <mx:columns>
         <mx:DataGridColumn dataField="Album"/>
         <mx:DataGridColumn dataField="Price"/>
      </mx:columns>
   </mx:DataGrid>
</s:Application>
```

In this example, you use the `intialize` event to set the `dataProvider` property of the DataGrid control.

In some situations, you might set the `dataProvider` property, and then immediately attempt to perform an action on the control based on the setting of the `dataProvider` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridValidateNowSelindex.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">
   <fx:Script>
       <![CDATA[
            import mx.collections.ArrayList;
            [Bindable]
            private var DGArray:ArrayList = new ArrayList([
                {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}]);

            // Initialize initDG ArrayList variable from the ArrayList.
            public function initData():void {
                myGrid.dataProvider = DGArray;
                myGrid.validateNow();
                myGrid.selectedIndex=1;
            }
        ]]>
   </fx:Script>
   <mx:DataGrid id="myGrid">
      <mx:columns>
         <mx:DataGridColumn dataField="Album"/>
         <mx:DataGridColumn dataField="Price"/>
      </mx:columns>
   </mx:DataGrid>
</s:Application>
```

In this example, setting the `selectedindex` to 1 might fail because the DataGrid control is in the process of setting the `dataProvider` property. Therefore, you insert the `validateNow()` method after setting the data provider. The `validateNow()` method validates and updates the properties and layout of the control, and then redraws it, if necessary.

Do not insert the `validateNow()` method every time you set the `dataProvider` property because it can affect the performance of your application; it is only required in some situations when you attempt to perform an operation on the control immediately after setting its `dataProvider` property.

Example: Using a collection

The following sample code shows how you can use a standard collection, an ArrayCollection object, to represent and manipulate an Array for use in a control. This example shows the following features:

- Using an ArrayCollection to represent data in an Array
- Sorting the ArrayCollection
- Inserting data in the ArrayCollection

Note that if the example did not include sorting, it could use an ArrayList rather than an ArrayCollection as the data provider.

This example also shows the insertion's effect on the Array and the ArrayCollection representation of the Array:

```
<?xml version="1.0"?>
<!-- dpcontrols\SimpleDP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" width="600"
    initialize="sortAC();">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import spark.collections.Sort;
            import spark.collections.SortField;

            // Function to sort the ArrayCollection in descending order.
            public function sortAC():void {
                var sortA:Sort = new Sort();
                sortA.fields=[new SortField("label")];
                myAC.sort=sortA;
                //Refresh the collection view to show the sort.
                myAC.refresh();
            }
            // Function to add an item in the ArrayCollection.
            // Data added to the view is also added to the underlying Array.
            // The ArrayCollection must be sorted for this to work.
            public function addItemToMyAC():void {
                myAC.addItem({label:"MD", data:"Annapolis"});
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- An ArrayCollection with an array of objects -->
        <mx:ArrayCollection id="myAC">
            <!-- Use an fx:Array tag to associate an id with the array. -->
            <fx:Array id="myArray">
                <fx:Object label="MI" data="Lansing"/>
                <fx:Object label="MO" data="Jefferson City"/>
                <fx:Object label="MA" data="Boston"/>
                <fx:Object label="MT" data="Helena"/>
                <fx:Object label="ME" data="Augusta"/>
                <fx:Object label="MS" data="Jackson"/>
                <fx:Object label="MN" data="Saint Paul"/>
            </fx:Array>
        </mx:ArrayCollection>
    </fx:Declarations>
    <s:HGroup width="100%">
        <!-- A ComboBox populated by the collection view of the Array. -->
        <s:ComboBox id="cb1" dataProvider="{myAC}"/>
        <s:Button id="b1" label="Add MD" click="addItemToMyAC();"/>
    </s:HGroup>
</s:Application>
```

Using simple data access properties and methods

Collections provide simple properties and methods for indexed access to linear data. These properties and methods are defined in the IList interface, which provides a direct representation of the underlying data object. Any operation that changes the collection also changes the data provider in a similar manner: if you insert an item as the third item in the collection, it is also the third item in the underlying data object, which is an Array when working with ArrayList or an ArrayCollection object. The underlying data object is an XMLList object when working with XMLListCollection objects.

Note: *If you use the ICollectionView interface to sort or filter a collection, do not use the IList interface to manipulate the data, because the results are indeterminate.*

Simple data access properties and methods let you do the following:

- Get, set, add, or remove an item at a specific index into the collection
- Add an item at the end of the collection
- Get the index of a specific item in the collection
- Remove all items in the collection
- Get the length of the collection

You can use this functionality directly on ArrayList, ArrayCollection, and XMLListCollection objects and also on the `dataProvider` property of any standard Flex data provider component.

The following sample code uses an ArrayList object to display an Array of elements in a ComboBox control. For an example that shows how to manage an ArrayCollection of objects with multiple fields, see “Example: Modifying data in a DataGrid control” on page 924.

In the following example the Array data source initially consists of the following elements:

```
"AZ", "MA", "MZ", "MN", "MO", "MS"
```

When you click the Button control, the application uses the `length` property of the ArrayList and several of its methods to do the following:

1. Change the data in the Array and the displayed data in the ComboBox control to a correct alphabetical list of the U.S. state abbreviations for states that start with M:

   ```
   MA, ME, MI, MN, MO, MS, MT
   ```

2. Display in a TextArea control information about the tasks it performed and the resulting Array.

The code includes comments that describe the changes to the data object.

```
<?xml version="1.0"?>
<!-- dpcontrols\UseIList.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.collections.*;
            // The data provider is an Array of Strings
            public var myArray:Array = ["AZ", "MA", "MZ", "MN", "MO", "MS"];
            // Declare an ArrayList that represents the Array.
            [Bindable]
            public var myAL:ArrayList;
            //Initialize the ArrayList.
            public function initData():void {
                myAL = new ArrayList(myArray);
            }
            // The function to change the collection, and therefore
            // the Array.
            public function changeCollection():void {
                // Get the original collection length.
                var oldLength:int=myAL.length;
                // Remove the invalid first item, AZ.
                var removedItem:String=String(myAL.removeItemAt(0));
                // Add ME as the second item. (ILists used 0-based indexing.)
                myAL.addItemAt("ME", 1);
                // Add MT at the end of the Array and collection.
                myAL.addItem("MT");
                // Change the third item from MZ to MI.
                myAL.setItemAt("MI", 2);
                // Get the updated collection length.
                var newLength:int=myAL.length;
                // Get the index of the item with the value ME.
                var addedItemIndex:int=myAL.getItemIndex("ME");
                // Get the fifth item in the collection.
                var index4Item:String=String(myAL.getItemAt(4));
                // Display the information in the TextArea control.
                ta1.text="Start Length: " + oldLength + ". New Length: " +
                    newLength;
                ta1.text+=".\nRemoved " + removedItem;
                ta1.text+=".\nAdded ME at index " + addedItemIndex;
                ta1.text+=".\nThe item at index 4 is " + index4Item + ".";
                // Show that the base Array has been changed.
                ta1.text+="\nThe base Array is: " + myArray.join();
            }
        ]]>
    </fx:Script>
    <s:ComboBox id="myCB" dataProvider="{myAL}"/>
    <s:TextArea id="ta1" height="75" width="300"/>
    <s:Button label="rearrange list" click="changeCollection();"/>
</s:Application>
```

Working with data views

Classes that implement the ICollectionView interface, such as ArrayCollection and XMLListCollection, provide a *view* of the underlying data object as a collection of items. Although they are more complex, data views give you more flexibility than the simple data access methods and properties that are defined in the IList interface. Data views provide the following features:

- You can modify the data view to show the data in sorted order or to show a subset of the items in the data provider without changing the underlying data. For more information, see "Sorting and filtering data for viewing" on page 912.
- You can access the collection data by using a *cursor*, which lets you move through the collection, use bookmarks to save specific locations in the collection, and insert and delete items in the collection (and therefore in the underlying data source). For more information, see "Using a view cursor" on page 914.
- You can represent remote data that might not initially be available, or parts of the data set that might become available at different times. For more information, see "Collection change notification" on page 923 and "Remote data in data provider components" on page 939.

You can use data views directly on ArrayCollection and XMLListCollection objects and also on the `dataProvider` property of standard Flex data provider components *except* those that are subclasses of the NavBar class (ButtonBar, LinkBar, TabBar, and ToggleButtonBar). It is a better practice to work directly on the collection objects than on the `dataProvider` property.

Note that the ArrayList class implements the IList interface but not the ICollectionView interface.

Sorting and filtering data for viewing

Collections let you sort and filter data in the data view so that the data in the collection is a reordered subset of the underlying data. Data view operations have no effect on the underlying data object content, only on the subset of data that the collection view represents, and therefore on what is displayed by any control that uses the collection.

Sorting

A Sort object lets you sort data in a collection. You can specify multiple fields to use in sorting the data, require that the resulting entries be unique, and specify a custom comparison function to use for ordering the sorted output. You can also use a Sort object to find items in a collection. When you create a Sort object, or change its properties, you must call the `refresh()` method on the collection to show the results.

You use SortField objects to specify the fields to use in the sort. You create SortField objects and put them in the Sort class object's `fields` array.

To create case-insensitive sorts, use the `ignoreCase` property of the SortingCollator.

Filtering

You use a filter function to limit the data view in the collection to a subset of the source data object. The function must take a single Object parameter, which corresponds to a collection item, and must return a Boolean value specifying whether to include the item in the view. As with sorting, when you specify or change the filter function, you must call the `refresh()` method on the collection to show the filtered results. To limit a collection view of an array of strings to contain only strings starting with M, for example, use the following filter function:

```
public function stateFilterFunc(item:Object):Boolean {
    return item >= "M" && item < "N";
}
```

Example: Sorting and filtering an ArrayCollection

The following example shows the use of the filter function and a sort together. You can use the buttons to sort the collection, to filter the collection, or to do both. Use the Reset button to restore the collection view to its original state.

```
<?xml version="1.0"?>
<!-- dpcontrols\SortFilterArrayCollection.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import spark.collections.Sort;
            import spark.collections.SortField;

            /* Function to sort the ICollectionView
               in ascending order. */
            public function sortAC():void {
                var sortA:Sort = new Sort();
                sortA.fields=[new SortField("label")];
                myAC.sort=sortA;
                //Refresh the collection view to show the sort.
                myAC.refresh();
            }
            /* Function to filter out all items with labels
               that are not in the range of M-N. */
            public function stateFilterFunc(item:Object):Boolean {
                return item.label >= "M" && item.label < "O";
            }

            /* Function to apply the filter function the ICollectionView. */
            public function filterAC():void {
                myAC.filterFunction=stateFilterFunc;
                /* Refresh the collection view to apply the filter. */
                myAC.refresh();
            }
            /* Function to Reset the view to its original state. */
            public function resetAC():void {
                myAC.filterFunction=null;
                myAC.sort=null;
                //Refresh the collection view.
                myAC.refresh();
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- An ArrayCollection with an array of objects. -->
        <mx:ArrayCollection id="myAC">
            <fx:Array id="myArray">
                <fx:Object label="LA" data="Baton Rouge"/>
                <fx:Object label="NH" data="Concord"/>
                <fx:Object label="TX" data="Austin"/>
                <fx:Object label="MA" data="Boston"/>
                <fx:Object label="AZ" data="Phoenix"/>
                <fx:Object label="OR" data="Salem"/>
                <fx:Object label="FL" data="Tallahassee"/>
                <fx:Object label="MN" data="Saint Paul"/>
                <fx:Object label="NY" data="Albany"/>
            </fx:Array>
        </mx:ArrayCollection>
    </fx:Declarations>
    <!-- Buttons to filter, sort, or reset the view in the second ComboBox
            control. -->
    <s:HGroup width="100%">
        <s:Button id="sortButton" label="Sort" click="sortAC();"/>
        <s:Button id="filterButton" label="Filter" click="filterAC();"/>
        <s:Button id="resetButton" label="Reset" click="resetAC();"/>
    </s:HGroup>
    <s:ComboBox id="cb1" dataProvider="{myAC}"/>
</s:Application>
```

For a more complex example of sorting a DataGrid control, which does both an initial sort of the data and a custom sort when you click a column heading, see “Create a custom sort for the Spark DataGrid control” on page 571.

Using a view cursor

You use a view cursor to traverse the items in a collection’s data view and to access and modify data in the collection. A *cursor* is a position indicator; it points to a particular item in the collection. Collections have a `createCursor()` method that returns a view cursor. View cursor methods and properties are defined in the IViewCursor interface.

You can use view cursor methods and properties to perform the following operations:

- Move the cursor backward or forward
- Move the cursor to specific items
- Get the item at a cursor location
- Add, remove, and change items
- Save a cursor position by using a bookmark, and return to it later

When you use standard Flex collection classes, ArrayCollection and XMLListCollection, you use the IViewCursor interface directly; as the following code snippet shows, you do not reference an object instance:

```
public var myAC:ICollectionView = new ArrayCollection(myArray);
public var myCursor:IViewCursor;
.
.
myCursor=myAC.createCursor();
```

Manipulating the view cursor

A view cursor object includes the following methods and properties for moving the cursor:

1 The `moveNext()` and `movePrevious()` methods move the cursor forward and backward by one item. Use the `beforeFirst` and `afterLast` properties to check whether you’ve reached the bounds of the view. The following example moves the cursor to the last item in the view:

```
while (! myCursor.afterLast) {
        myCursor.moveNext();
}
```

2 The `findAny()`, `findFirst()`, and `findLast()` methods move the cursor to an item that matches the parameter. Before you can use these methods, you must apply a Sort to the collection (because the functions use Sort methods).

If it is not important to find the first occurrence of an item or the last occurrence of an item in a nonunique index, the `findAny()` method can be somewhat more efficient than either the `findFirst()` or the `findLast()` method.

If the associated data is from a remote source, and not all of the items are cached locally, the find methods begin an asynchronous fetch from the remote source; if a fetch is already in progress, they wait for it to complete before making another fetch request.

The following example finds an item inside a collection of simple objects—in this case, an ArrayCollection of state ZIP code strings. It creates a default Sort object, applies it to an ArrayCollection object, and finds the first instance of the string `"MZ"` in a simple array of strings:

```
var sortD:Sort = new Sort();
// The null first parameter on the SortField constructor specifies a
// collection of simple objects (String, numeric, or Boolean values).
// The true second parameter specifies a case-insensitive sort.
sortD.fields = [new SortField(null, true)];
myAC.sort=sortD;
myAC.refresh();
myCursor.findFirst("MZ");
```

To find a complex object, you can use the `findFirst()` method to search on multiple sort fields. You cannot, however, skip fields in the parameter of any of the find methods. If an object has three fields, for example, you can specify any of the following field combinations in the parameter: 1, 1,2, or 1,2,3, but you cannot specify only fields 1 and 3.

Both of the following lines find an object with the label value `"ME"` and data value `"Augusta"`:

```
myCursor.findFirst({label:"ME"});
myCursor.findFirst({label:"ME", data:"Augusta"});
```

❖ The `seek()` method moves the cursor to a position relative to a bookmark. You use this method to move the cursor to the first or last item in a view, or to move to a bookmark position that you have saved.

Getting, adding, and removing data items

A view cursor object includes the following methods and properties for accessing and changing data in the view:

- The `current` property is a reference to the item at the current cursor location.
- The `insert()` method inserts an item before the current cursor location. However, if the collection is sorted (for example, to do a `find()` operation, the sort moves the item to the sorted order location, not to the cursor location.
- The `remove()` method removes the item at the current cursor location; if the removed item is not the last item, the cursor points to the location after the removed item.

The following example shows the results of using `insert()` and `remove()` on the `current` property:

```
<?xml version="1.0"?>
<!-- dpcontrols\GetAddRemoveItems.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.*;
            public var myArray:Array = [{label:"MA", data:"Massachusetts"},
            {label:"MN", data:"Minnesota"}, {label:"MO", data:"Missouri"}];
            [Bindable]
            public var myAC:ArrayCollection;
            public var myCursor:IViewCursor;
            /* Initialize the ArrayCollection when you
               initialize the application. */
            public function initData():void {
                myAC = new ArrayCollection(myArray);
            }
            /* The function to change the collection,
               and therefore the Array. */
            public function testCollection():void {
                /* Get an IViewCursor object for accessing the collection data. */
                myCursor=myAC.createCursor();
                ta1.text="At start, the cursor is at: " + myCursor.current.label + ".";
                var removedItem:String=String(myCursor.remove());
                ta1.text+="\nAfter removing the current item, the cursor is at: "
                    + myCursor.current.label + ".";
                myCursor.insert({label:"ME", data:"Augusta"});
                ta1.text+="\nAfter adding an item, the cursor is at: "
                    + myCursor.current.label + ".";
            }
        ]]>
    </fx:Script>
    <s:ComboBox id="myCB" dataProvider="{myAC}"/>
    <s:TextArea id="ta1" height="75" width="350"/>
    <s:Button label="Run Test" click="testCollection();"/>
</s:Application>
```

Using bookmarks

You use a bookmark to save a cursor location for later use. You can also use the built-in `FIRST` and `LAST` bookmark properties to move the cursor to the first or last item in the data view.

Create and use a bookmark

1 Move the cursor to a desired location in the data view.

2 Assign the current value of the `bookmark` property to a variable, as in the following line:

```
var myBookmark:CursorBookmark=myCursor.bookmark;
```

3 Do some operations that might move the cursor.

4 When you must return to the bookmarked cursor location (or to a specific offset from the bookmarked location), call the IViewCursor `seek()` method, as in the following line:

```
myCursor.seek(myBookmark);
```

The following example counts the number of items in a collection between the selected item in a ComboBox control and the end of the collection, and then returns the cursor to the initial location:

```
<?xml version="1.0"?>
<!-- dpcontrols\UseBookmarks.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="run();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.collections.IViewCursor;
            import mx.collections.CursorBookmark;
            import spark.collections.Sort;
            import spark.collections.SortField;
            private var myCursor:IViewCursor;
            // Initialize variables.
            public function run():void {
                // Initialize the cursor.
                myCursor=myAC.createCursor();
                // The findFirst() method, used in
                // countFromSelection() requires a
                // sorted view.
                var sort:Sort = new Sort();
                sort.fields=[new SortField("label")];
                myAC.sort=sort;
                //You must refresh the view to apply the sort.
                myAC.refresh();
            }
            // Count the items following the current
            // cursor location.
            public function countLast(theCursor:IViewCursor):int {
                var counter:int=0;
                // Set a bookmark at the current cursor location.
                var mark:CursorBookmark=theCursor.bookmark;
                // Move the cursor to the end of the Array.
                // The moveNext() method returns false when the cursor
                // is after the last item.
                    while (theCursor.moveNext()) {
                    counter++;
                }
                // Return the cursor to the initial location.
                theCursor.seek(mark);
                return counter;
            }
            // Function triggered by ComboBox change event.
            // Calls the countLast() function to count the
            // number of items to the end of the collection.
            public function countFromSelection():void {
                myCursor.findFirst(myCB.selectedItem);
                var count:int = countLast(myCursor);
                ta1.text += myCursor.current.label + " is " + count +
                                " from the last item.\n";
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- The data provider, an ArrayCollection with an array of objects. -->
        <mx:ArrayCollection id="myAC">
            <fx:Object label="MA" data="Boston"/>
            <fx:Object label="ME" data="Augusta"/>
            <fx:Object label="MI" data="Lansing"/>
            <fx:Object label="MN" data="Saint Paul"/>
            <fx:Object label="MO" data="Jefferson City"/>
            <fx:Object label="MS" data="Jackson"/>
            <fx:Object label="MT" data="Helena"/>
        </mx:ArrayCollection>
    </fx:Declarations>
    <s:ComboBox id="myCB"
        dataProvider="{myAC}" change="countFromSelection();"/>
    <s:TextArea id="ta1" height="200" width="175"/>
</s:Application>
```

Example: Updating an Array by using data view methods and properties

The following example uses the data view methods and properties of an ArrayCollection object to display an Array with the following elements in a ComboBox control:

```
"AZ", "MA", "MZ", "MN", "MO", "MS"
```

When you click the Update View button, the application uses the `length` property and several methods of the ICollectionView interface to do the following:

- Change the data in the array and the displayed data in the ComboBox control to a correct alphabetical list of the U.S. state abbreviations for the following states that start with the letter M:

  ```
  MA, ME, MI, MN, MO, MS, MT
  ```

- Save a bookmark that points to the ME item that it adds, and later restores the cursor to this position.
- Display in a TextArea control information about the tasks it performed and the resulting array.

When you click the Sort button, the application reverses the order of the items in the view, and limits the viewed range to ME–MO.

When you click the Reset button, the application resets the data provider array and the collection view.

```
<?xml version="1.0"?>
<!-- dpcontrols\UpdateArrayViaICollectionView.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.collections.CursorBookmark;
            import mx.collections.IViewCursor;
            import spark.collections.Sort;
            import spark.collections.SortField;

            // The data provider is an array of Strings.
            public var myArray:Array = ["AZ", "MA", "MZ", "MN", "MO", "MS"];
            // Declare an ArrayCollection that represents the Array.
            // The variable must be bindable so the ComboBox can update properly.
            [Bindable]
            public var myAC:ArrayCollection;
            //Boolean flag to ensure the update routine hasn't been run before.
            public var runBefore:Boolean=false;
            //Initialize the ArrayCollection the application initializes.
            public function initData():void {
                myAC = new ArrayCollection(myArray);
            }
            // The function to change the collection.
            public function changeCollection():void {
                //Running this twice without resetting causes an error.
                if (! runBefore) {
                    runBefore=true;
                    // Get an IViewCursor object for accessing the collection data.
                    var myCursor:IViewCursor=myAC.createCursor();
                    // Get the original collection length.
                    var oldLength:int=myAC.length;
                    // The cursor is initially at the first item; delete it.
                    var removedItem:String=String(myCursor.remove());
                    // Add ME as the second item.
                    // The cursor is at the (new) first item;
                    // move it to the second item.
                    myCursor.moveNext();
                    // Insert ME before the second item.
                    myCursor.insert("ME");
                    // Add MT at the end of the collection.
                    //Use the LAST bookmark property to go to the end of the view.
                    // Add an offset of 1 to position the cursor after the last item.
                    myCursor.seek(CursorBookmark.LAST, 1);
                    myCursor.insert("MT");
                    // Change MZ to MI.
                    // The findFirst() method requires a sorted view.
                    var sort:Sort = new Sort();
                    myAC.sort=sort;
                    // Refresh the collection view to apply the sort.
                    myAC.refresh();
                    // Make sure there is a MZ item, and no MI in the array.
                     if (myCursor.findFirst("MZ") && !myCursor.findFirst("MI")) {
                        // The IViewCursor does not have a replace operation.
                        // First, remove "MZ".
                        myCursor.remove();
                        // Because the view is now sorted, the insert puts this item
                        // in the right place in the sorted view, but at the end of
                        // the underlying Array data provider.
                        myCursor.insert("MI");
                    }
                    // Get the updated collection length.
                    var newLength:int=myAC.length;
                    // Set a bookmark at the item with the value ME,
                    myCursor.findFirst("ME");
                    var MEMark:CursorBookmark=myCursor.bookmark;
                    // Move the cursor to the last item in the Array.
                    myCursor.seek(CursorBookmark.LAST);
                    // Get the last item in the collection.
                    var lastItem:String=String(myCursor.current);
                    // Return the cursor to the bookmark position.
                    myCursor.seek(MEMark);
                    // Get the item at the cursor location.
                    var MEItem:String=String(myCursor.current);
                    // Display the information in the TextArea control.
                    ta1.text="Start Length: " + oldLength + ". End Length: "
                                    + newLength;
                    ta1.text+=".\nRemoved " + removedItem;
                    ta1.text+=".\nLast Item is " + lastItem;
                    ta1.text+=".\nItem at MEMark is " + MEItem;
                    // Show that the base Array has been changed.
                    // Notice that the Array is NOT in sorted order.
                    ta1.text+="\nThe base Array is: " + myArray.join();
                } // End runBefore condition
            }
            // Filter function used in the sortICV method to limit the range.
            public function MEMOFilter(item:Object):Boolean {
                return item >= "ME" && item <= "MO";
            }

            // Sort the collection view in descending order,
            // and limit the items to the range ME - MO.
            public function sortICV():void {
                var sort:Sort = new Sort();
                sort.fields=[new SortField(null, false, true)];
                myAC.filterFunction=MEMOFilter;
                myAC.sort=sort;
                // Refresh the ArrayCollection to apply the sort and filter
                // function.
                myAC.refresh();
                //Call the ComboBox selectedIndex() method to replace the "MA"
                //in the display with the first item in the sorted view.
```

```
                myCB.selectedIndex=0;
                ta1.text="Sorted";
            }
            //Reset the Array and update the display to run the example again.
            public function resetView():void {
                myArray = ["AZ", "MA", "MZ", "MN", "MO", "MS"];
                myAC = new ArrayCollection(myArray);
                ta1.text="Reset";
                runBefore=false;
            }
        ]]>
    </fx:Script>
    <s:ComboBox id="myCB" dataProvider="{myAC}"/>
    <s:TextArea id="ta1" height="75" width="300"/>
    <s:HGroup>
        <s:Button label="Update View" click="changeCollection();"/>
        <s:Button label="Sort View" click="sortICV();"/>
        <s:Button label="Reset View" click="resetView();"/>
    </s:HGroup>
</s:Application>
```

Collection events and manual change notification

Collections use events to indicate changes to the collection. You can use these events to monitor changes and update the display accordingly.

Collection events

Collections use CollectionEvent, PropertyChangeEvent, and FlexEvent objects in the following ways:

- Collections dispatch a CollectionEvent (mx.events.CollectionEvent) event whenever the collection changes. All collection events have the `type` property value `CollectionEvent.COLLECTION_CHANGE`.
- The CollectionEvent object includes a `kind` property that indicates the way in which the collection changed. You can determine the change by comparing the `kind` property value with the CollectionEventKind constants; for example, `UPDATE`.
- The CollectionEvent object includes an `items` property that is an Array of objects whose type varies depending on the event kind. For ADD and REMOVE kind events, the array contains the added or removed items. For UPDATE events, the `items` property contains an Array of PropertyChangeEvent event objects. This object’s properties indicate the type of change and the property value before and after the change.
- The `PropertyChangeEvent` class `kind` property indicates the way in which the property changed. You can determine the change type by comparing the `kind` property value with the PropertyChangeEventKind constants; for example, `UPDATE`.
- View cursor objects dispatch a FlexEvent class event with the `type` property value of `mx.events.FlexEvent.CURSOR_UPDATE` when the cursor position changes.

You use collection events to monitor changes to a collection to update the display. For example, if a custom control uses a collection as its data provider, and you want the control to be updated dynamically and to display the revised data each time the collection changes, the control can monitor the collection events and update accordingly.

You could, for example, build a simple rental-car reservation system that uses collection events. This application uses `COLLECTION_CHANGE` event type listeners for changes to its `reservations` and `cars` data collections.

The CollectionEvent listener method, named `reservationsChanged`, tests the event `kind` field and does the following:

- If the event `kind` property is `ADD`, iterates through the objects in the event's `items` property and calls a function to update the reservation information display with boxes that display the time span of each reservation.
- If the event `kind` property is `REMOVE`, iterates through the objects in the event's `items` property and calls a function to remove the reservation box for each item.
- If the event `kind` property is `UPDATE`, iterates through the `PropertyChangeEvent` objects in the event's `items` property and calls the update function to update each item.
- If the event `kind` property is `RESET`, calls a function to reset the reservation information.

The following example shows the `reservationsChanged` CollectionEvent event listener function:

```
private function reservationsChanged(event:CollectionEvent):void {
    switch (event.kind) {
        case CollectionEventKind.ADD:
            for (var i:uint = 0; i < event.items.length; i++) {
                    updateReservationBox(Reservation(event.items[i]));
            }
            break;

        case CollectionEventKind.REMOVE:
            for (var i:uint = 0; i < event.items.length; i++) {
                removeReservationBox(Reservation(event.items[i]));
            }
            break;

        case CollectionEventKind.UPDATE:
            for (var i:uint = 0; i < event.items.length; i++) {
                if (event.items[i] is PropertyChangeEvent) {
                    if (PropertyChangeEvent(event.items[i]) != null) {
                         updateReservationBox(Reservation(PropertyChangeEvent(
                            event.items[i]).source));
                    }
                }
                else if (event.items[i] is Reservation) {
                    updateReservationBox(Reservation(event.items[i]));
                }
        }
        break;

        case CollectionEventKind.RESET:
            refreshReservations();
            break;
    }
}
```

The `updateReservationBox()` method either shows or hides a box that shows the time span of the reservation. The `removeReservationBox()` method removes a reservation box. The `refreshReservations()` method redisplays all current reservation information.

For more information on the application and the individual methods, see the sample code.

Collection change notification

Collections include the itemUpdated() method, which notifies a collection that the underlying data has changed and ensures that the collection's data view is up to date when items in the underlying data object do not implement the IEventDispatcher interface. This method takes the item that was modified, the property in the item that was updated, and its old and new values as parameters. Collections also provide the `enableAutoUpdate()` and `disableAutoUpdate()` methods, which enable and disable the automatic updating of the data view when the underlying data provider changes.

Using the itemUpdated() method

Use the `itemUpdated()` method to notify the collection of changes to a data provider object if the object does not implement the IEventDispatcher interface; in this case the object is not monitorable. Adobe Flash and Flex Objects and other basic data types do not implement this interface. Therefore you must use the `itemUpdated()` method to update the collection when you modify the properties of a data provider such as an Array or through the display object.

You can also use the `itemUpdated()` method if you must use an Array, rather than a collection, as an MX control's data provider. Then the component wraps the Array in a collection wrapper. The wrapper must be manually notified of any changes made to the underlying Array data object, and you can use the `itemUpdated()method` for that notification.

You do *not* have to use the `itemUpdated()` method if you add or remove items directly in a collection or use any of the ICollectionView or IList methods to modify the collection.

Also, specifying the `[Bindable]` metadata tag above a class definition, or above a variable declaration within the class, ensures that the class implements the IEventDispatcher interface, and causes the class to dispatch propertyChange events. If you specify the `[Bindable]` tag above the class declaration, the class dispatches propertyChange events for all properties; if you mark only specific properties as `[Bindable]`, the class dispatches events for only those properties. The collection listens for the propertyChange events. Therefore, if you have a collection called `myCollection` that consists of instances of a class that has a `[Bindable]myVariable` variable, an expression such as `myCollection.getItemAt(0).myVariable="myText"` causes the item to dispatch an event, and you do not have to use the `itemUpdated()` method. (For more information on the `[Bindable]` metadata tag and its use, see "Data binding" on page 299.)

The most common use of the `itemUpdate()` method is to notify a collection of changes to a custom class data source that you cannot make bindable or modify to implement the IEventDispatcher interface. The following schematic example shows how you could use the `itemUpdated()` method in such a circumstance.

Assume you have a class that you do not control or edit and that looks like the following:

```
public class ClassICantEdit {
    public var field1:String;
    public var field2:String;
}
```

You have an ArrayCollection that uses these objects, such as the following, which you populate with `classICantEdit` objects:

```
public var myCollection:ArrayCollection = new ArrayCollection();
```

You have a DataGrid control such as the following:

```
<s:DataGrid dataProvider="{myCollection}"/>
```

When you update a field in the `myCollection` ArrayCollection, as follows, the DataGrid control is not automatically updated:

```
myCollection.getItemAt(0).field1="someOtherValue";
```

To update the DataGrid control, you must use the collection's `itemUpdated()` method:

```
myCollection.itemUpdated(collectionOfThoseClasses.getItemAt(0));
```

Disabling and enabling automatic updating

A collection's disableAutoUpdate() method prevents events that represent changes to the underlying data from being broadcast by the view. It also prevents the collection from being updated as a result of these changes.

Use this method to prevent the collection, and therefore the control that uses it as a data provider, from showing intermediate changes in a set of multiple changes. The DataGrid class, for example, uses the `disableAutoUpdate()` method to prevent updates to the collection while a specific item is selected. When the item is no longer selected, the DataGrid control calls the `enableAutoUpdate()` method. Doing this ensures that, if a DataGrid control uses a sorted collection view, items that you edit do not jump around while you're editing.

You can also use the `disableAutoUpdate()` method to optimize performance in cases where multiple items in a collection are being edited at once. By disabling the auto update until all changes are made, a control like the DataGrid control can receive an update event as a single batch instead of reacting to multiple events.

The following code snippet shows the use of the `disableAutoUpdate()` and `enableAutoUpdate()` methods:

```
var obj:myObject = myCollection.getItemAt(0);
myCollection.disableAutoUpdate();
obj.prop1 = 'foo';
obj.prop2 = 'bar';
myCollection.enableAutoUpdate();
```

Example: Modifying data in a DataGrid control

The following example lets you add, remove, or modify data in a DataGrid control:

```
<?xml version="1.0"?>
<!-- dpcontrols\ModifyDataGridData.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="500" height="600" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.*;
            import mx.collections.*;

            // Add event information to a log (displayed in the TextArea).
            public function collectionEventHandler(event:CollectionEvent):void {
                switch(event.kind) {
                    case CollectionEventKind.ADD:
                        addLog("Item "+ event.location + " added");
                        break;
                    case CollectionEventKind.REMOVE:
                        addLog("Item "+ event.location + " removed");
                        break;
                    case CollectionEventKind.REPLACE:
                        addLog("Item "+ event.location + " Replaced");
                        break;
                    case CollectionEventKind.UPDATE:
```

```
                    addLog("Item updated");
                    break;
            }
        }
        // Helper function for adding information to the log.
        public function addLog(str:String):void {
            log.text += str + "\n";
        }

        // Add a person to the ArrayCollection.
        public function addPerson():void {
            ac.addItem({first:firstInput.text, last:lastInput.text,
                email:emailInput.text});
                clearInputs();
        }

        // Remove a person from the ArrayCollection.
        public function removePerson():void {
            // Make sure an item is selected.
            if (dg.selectedIndex >= 0) {
                ac.removeItemAt(dg.selectedIndex);
        }
    }

    // Update an existing person in the ArrayCollection.
    public function updatePerson():void {
        // Make sure an item is selected.
        if (dg.selectedItem !== null) {
            ac.setItemAt({first:firstInput.text, last:lastInput.text,
                email:emailInput.text}, dg.selectedIndex);
        }
    }

    // The change event listener for the DataGrid.
    // Clears the text input controls and updates them with the contents
    // of the selected item.
    public function dgChangeHandler():void {
        clearInputs();
        firstInput.text = dg.selectedItem.first;
        lastInput.text = dg.selectedItem.last;
        emailInput.text = dg.selectedItem.email;
    }

    // Clear the text from the input controls.
    public function clearInputs():void {
        firstInput.text = "";
        lastInput.text = "";
        emailInput.text = "";
    }
    // The labelFunction for the ComboBox;
    // Puts first and last names in the ComboBox.
    public function myLabelFunc(item:Object):String {
        return item.first + " " + item.last;
    }
    ]]>
</fx:Script>
```

```
    <fx:Declarations>
        <!-- The ArrayCollection used by the DataGrid and ComboBox. -->
        <mx:ArrayCollection id="ac"
            collectionChange="collectionEventHandler(event)">
            <mx:source>
                <fx:Object first="Matt" last="Matthews" email="matt@myco.com"/>
                <fx:Object first="Sue" last="Sanderson" email="sue@myco.com"/>
                <fx:Object first="Harry" last="Harrison" email="harry@myco.com"/>
            </mx:source>
        </mx:ArrayCollection>
    </fx:Declarations>
    <mx:DataGrid width="450" id="dg" dataProvider="{ac}"
            change="dgChangeHandler()">
        <mx:columns>
            <mx:DataGridColumn dataField="first" headerText="First Name"/>
            <mx:DataGridColumn dataField="last" headerText="Last Name"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
    <!-- The ComboBox and DataGrid controls share an ArrayCollection as their
        data provider.
        The ComboBox control uses the labelFunction property to construct the
        labels from the dataProvider fields. -->
    <s:ComboBox id="cb" dataProvider="{ac}" labelFunction="myLabelFunc"/>

    <!-- Form for data to add or change in the ArrayCollection. -->
    <s:Form>
       <s:FormItem label="First Name">
            <s:TextInput id="firstInput"/>
       </s:FormItem>
       <s:FormItem label="Last Name">
            <s:TextInput id="lastInput"/>
       </s:FormItem>
       <s:FormItem label="Email">
            <s:TextInput id="emailInput"/>
       </s:FormItem>
    </s:Form>

    <s:HGroup>
        <!-- Buttons to initiate operations on the collection. -->
        <s:Button label="Add New" click="addPerson()"/>
        <s:Button label="Update Selected" click="updatePerson()"/>
        <s:Button label="Remove Selected" click="removePerson()"/>
        <!-- Clear the text input fields. -->
        <s:Button label="Clear" click="clearInputs()"/>
    </s:HGroup>

    <!-- The application displays event information here -->
    <s:Label text="Log"/>
    <s:TextArea id="log" width="100" height="100%"/>
</s:Application>
```

Hierarchical data objects

You use hierarchical data objects with the controls that display a nested hierarchy of nodes and subnodes, such as tree branches and leaves, as well as Menu submenus and items. The following controls use hierarchical data objects:

- Menu
- MenuBar
- PopUpMenuButton
- Tree

The hierarchical components all use the same mechanism to work with the data provider. The following examples use the Tree control, but the examples apply to the other components.

About hierarchical data objects

The Flex framework, by default, supports two types of hierarchical data objects.

XML can be any of the following: Strings containing well-formed XML; or XML, XMLList, or XMLListCollection objects, including objects generated by the <fx:XML> and <fx:XMLList> compile-time tags. (These tags support data binding, which you cannot do directly in ActionScript.) Flex can automatically structure a Tree or menu-based control to reflect the nesting hierarchy of well-formed XML.

Objects can be any set of nested Objects or Object subclasses (including Arrays or ArrayCollection objects) that have a structure where the children of a node are in a `children` field. For more information, see "Creating a custom data descriptor" on page 931. You can also use the `<fx:Model>` compile-time tag to create nested objects that support data binding, but you must follow the structure defined in "Using the <fx:Model> tag with Tree and menu-based controls" on page 929.

You can add support for other hierarchical data provider structures, such as nested Objects where the children might be in fields with varying names.

Data descriptors and hierarchical data structure

Hierarchical data used in Tree and menu-based controls must be in a form that can be parsed and manipulated by using a data descriptor class. A *data descriptor* is a class that provides an interface between the hierarchical control and the data provider object. It implements a set of control-specific methods to determine the data provider contents and structure; to get, add, and remove data; and to change control-specific data properties.

Flex defines two data descriptor interfaces for hierarchical controls:

- ITreeDataDescriptor — Methods used by Tree controls
- IMenuDataDescriptor — Methods for Menu, MenuBar, and PopUpMenuButton controls

The Flex framework provides a DefaultDataDescriptor class that implements both interfaces. You can use the `dataDescriptor` property to specify a custom data descriptor class that handles data models that do not conform to the default descriptor structure.

Data descriptor methods and source requirements

The following table describes the methods of both interfaces, and the behavior of the DefaultDataDescriptor class. The first line of each interface/method entry indicates whether the method belongs to the ITreeDataDescriptor interface, the IMenuDataDescriptor interface, or both interfaces, and therefore indicates whether the method is used for trees, menus, or both.

Method	**Returns**	**DefaultDataDescriptor behavior**
hasChildren(*node*, [*model*])	A Boolean value indicating whether the node is a branch with children.	For XML, returns `true` if the node has at least one child element. For other objects, returns `true` if the node has a nonempty `children` field.
getChildren(*node*, [*collection*])	A node's children.	For XML, returns an XMLListCollection with the child elements. For other Objects, returns the contents of the node's `children` field.
isBranch(*node*, [*collection*])	Whether a node is a branch.	For XML, returns `true` if the node has at least one child, or if it has an `isBranch` attribute. For other Objects, returns `true` if the node has an `isBranch` field.
getData(*node*, [*collection*])	The node data.	Returns the node.
addChildAt(*node*, *child*, *index*, [*model*])	A Boolean value indicating whether the operation succeeded.	For all cases, inserts the node as a child object before the node currently in the index location.
removeChildAt (*node*, *index*, [*model*])	A Boolean value indicating whether the operation succeeded.	For all cases, removes the child of the node in the index location.
getType(*node*) (IMenuDataDescriptor only)	A String with the menu node type. Meaningful values are `check`, `radio`, and `separator`.	For XML, returns the value of the `type` attribute of the node. For other Objects, returns the contents of the node's `type` field.
isEnabled(*node*) (IMenuDataDescriptor only)	A Boolean value indicating whether a menu node is enabled.	For XML, returns the value of the `enabled` attribute of the node. For other Objects, returns the contents of the node's `enabled` field.
setEnabled(*node*, *value*) (IMenuDataDescriptor only)		For XML, sets the value of the `enabled` attribute of the node to `true` or `false`. For other Objects, sets the contents of the node's `enabled` field.
isToggled(*node*) (IMenuDataDescriptor only)	A Boolean value indicating whether a menu node is selected	Returns the value of the node's `toggled` attribute.
setToggled(*node*, *value*) (IMenuDataDescriptor only)		For XML, sets the value of the `selected` attribute of the node to `true` or `false`. For other Objects, sets the contents of the node's `enabled` field.
getGroupName(*node*) (IMenuDataDescriptor only)	The name of the radio button group to which the node belongs.	For XML, returns the value of the `groupName` attribute of the node. For other Objects, returns the contents of the node's `groupName` field.

The following example Object follows the default data provider structure for a Tree control, and is correctly handled by the DefaultDataDescriptor class:

```
[Bindable]
public var fileSystemStructure:Object =
    {label:"mx", children: [
        {label:"Containers", children: [
            {label:"Accordian", children:[]},
            {label:"DividedBox", children: [
                {label:"BoxDivider.as", data:"BoxDivider.as"},
                {label:"BoxUniter.as", data:"BoxUniter.as"}]},
            {label: "Grid", children:[]}]},
        {label: "Controls", children: [
            {label: "Alert", data: "Alert.as"},
            {label: "Styles", children: [
                {label: "AlertForm.as", data:"AlertForm.as"}]},
            {label: "Tree", data: "Tree.as"},
            {label: "Button", data: "Button.as"}]},
        {label: "Core", children:[]}
    ]};
```

For objects, the root is the Object instance, so there must always be a single root (as with XML). You could also use an Array containing nested Arrays as the data provider. In this case the provider has no root; each element in the top level array appears at the top level of the control.

The DefaultDataDescriptor can properly handle well-formed XML nodes. The `isBranch()` method, however, returns `true` only if the parameter node has child nodes or if the node has an `isBranch` attribute with the value `true`. Therefore, if your XML object uses any technique other than a `trueisBranch` attribute to indicate empty branches, you must create a custom data descriptor.

The DefaultDataDescriptor handles collections properly. For example, if a node's `children` property is an ICollectionView instance, the `getChildren()` method returns the children as an ICollectionView object.

Using the <fx:Model> tag with Tree and menu-based controls

The `<fx:Model>` tag lets you define a data provider structure in MXML. The Flex compiler converts the contents of the tag into a hierarchical graph of ActionScript Objects. The `<fx:Model>` tag has two advantages over defining an Object data provider in ActionScript:

- You can define the structure by using an easily read, XML-like format.
- You can bind structure entries to ActionScript variables, so that you can use `<fx:Model>` to create an object-based data provider that gets its data from multiple dynamic sources.

To use an `<fx:Model>` tag with a control that uses a data descriptor, the object generated by the compiler must conform to the data descriptor requirements, as discussed in "Data descriptors and hierarchical data structure" on page 927. Also, as with an XML object, the tag must have a single root element.

In most situations, you should consider using an <fx:XML> or <fx:XMLList> tag, as described in "XML-based data objects" on page 936, instead of using an `<fx:Model>` tag. The XML-based tags support data binding to elements, and the DefaultDataDescriptor class supports all well-structured XML. Therefore you can use a more natural structure, where node names can represent their function, and you do not have to artificially name nodes "children."

To use an `<fx:Model>` tag as the data provider for a control that uses the DefaultDataDescriptor class, all child nodes must be named "children." This requirement differs from the structure that you use with an Object, where the array that contains the child objects is named "children".

The following example shows the use of an `<fx:Model>` tag with data binding as a data provider for a menu, and shows how you can change the menu structure dynamically:

```
<?xml version="1.0"?>
<!-- dpcontrols\ModelWithMenu.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns="*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.controls.Menu;
            public var productMenu:Menu;
            public function initMenu(): void {
                productMenu = Menu.createMenu(null, Products.Department);
                productMenu.setStyle("disabledColor", 0xCC3366);
                productMenu.show(10,10);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:Model id="Products">
            <Root>
                <Department label="Toys">
                    <children label="Teddy Bears"/>
                    <children label="Action Figures"/>
                    <children label="Building Blocks"/>
                </Department>
                <Department label="Kitchen">
                    <children label="Electronics">
                        <children label="Crock Pot"/>
                        <children label="Panini Grill"/>
                    </children>
                    <children label="Cookware">
                        <children label="Grill Pan"/>
                        <children label="Iron Skillet" enabled="false"/>
                    </children>
                </Department>
                <!-- The items in this entry are bound to the form data -->
                <Department label="{menuName.text}">
                    <children label="{item1.text}"/>
                    <children label="{item2.text}"/>
                    <children label="{item3.text}"/>
                </Department>
            </Root>
        </fx:Model>
    </fx:Declarations>
    <s:Button label="Show Products" click="initMenu()"/>
    <!-- If you change the contents of the form, the next time you
        display the Menu, it will show the updated data in the last
        main menu item. -->
    <s:Form>
        <s:FormItem label="Third Submenu title">
            <s:TextInput id="menuName" text="Clothing"/>
        </s:FormItem>
        <s:FormItem label="Item 1">
            <s:TextInput id="item1" text="Sweaters"/>
        </s:FormItem>
        <s:FormItem label="Item 2">
            <s:TextInput id="item2" text="Shoes"/>
        </s:FormItem>
        <s:FormItem label="Item 3">
            <s:TextInput id="item3" text="Jackets"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Creating a custom data descriptor

If your hierarchical data does not fit the formats supported by the DefaultDataDescriptor class—for example, if your data is in an object that does not use a children field—you can write a custom data descriptor and specify it in your Tree control's `dataDescriptor` property. The custom data descriptor must implement all methods of the `ITreeDataDescriptor` interface.

The following example shows how you can create a custom data descriptor—in this case, for use with a Tree control. This data descriptor correctly handles a data provider that consists of nested ArrayCollection objects.

The following code shows the MyCustomTreeDataDescriptor class, which implements only the ITreeDataDescriptor interface, so it supports Tree controls but not menu-based controls. The custom class supports tree nodes whose children field is either an ArrayCollection or an Object. When getting a node's children, if the children object is an ArrayCollection, it returns the object; otherwise, it wraps the children object in an ArrayCollection before returning it. When adding a node, it uses a different method to add the node, depending on the children field type.

```
package myComponents
// myComponents/MyCustomTreeDataDescriptor.as
{
import mx.collections.ArrayCollection;
import mx.collections.CursorBookmark;
import mx.collections.ICollectionView;
import mx.collections.IViewCursor;
import mx.events.CollectionEvent;
import mx.events.CollectionEventKind;
import mx.controls.treeClasses.*;
public class MyCustomTreeDataDescriptor implements ITreeDataDescriptor
{
    // The getChildren method requires the node to be an Object
    // with a children field.
    // If the field contains an ArrayCollection, it returns the field
    // Otherwise, it wraps the field in an ArrayCollection.
    public function getChildren(node:Object,
        model:Object=null):ICollectionView
    {
        try
        {
            if (node is Object) {
                if(node.children is ArrayCollection){
                    return node.children;
                }else{
                    return new ArrayCollection(node.children);
                }
            }
        }
        catch (e:Error) {
            trace("[Descriptor] exception checking for getChildren");
        }
        return null;
    }
    // The isBranch method simply returns true if the node is an
    // Object with a children field.
    // It does not support empty branches, but does support null children
    // fields.
    public function isBranch(node:Object, model:Object=null):Boolean {
        try {
            if (node is Object) {
                if (node.children != null)  {
                    return true;
                }
            }
        }
        catch (e:Error) {
            trace("[Descriptor] exception checking for isBranch");
        }
        return false;
    }
    // The hasChildren method Returns true if the
    // node actually has children.
    public function hasChildren(node:Object, model:Object=null):Boolean {
        if (node == null)
            return false;
        var children:ICollectionView = getChildren(node, model);
        try {
            if (children.length > 0)
                return true;
        }
        catch (e:Error) {
        }
        return false;
    }
    // The getData method simply returns the node as an Object.
    public function getData(node:Object, model:Object=null):Object {
        try {
            return node;
        }
        catch (e:Error) {
        }
        return null;
    }
    // The addChildAt method does the following:
    // If the parent parameter is null or undefined, inserts
    // the child parameter as the first child of the model parameter.
    // If the parent parameter is an Object and has a children field,
    // adds the child parameter to it at the index parameter location.
    // It does not add a child to a terminal node if it does not have
    // a children field.
    public function addChildAt(parent:Object, child:Object, index:int,
            model:Object=null):Boolean {
        var event:CollectionEvent = new CollectionEvent(CollectionEvent.COLLECTION_CHANGE);
        event.kind = CollectionEventKind.ADD;
        event.items = [child];
        event.location = index;
        if (!parent) {
            var iterator:IViewCursor = model.createCursor();
            iterator.seek(CursorBookmark.FIRST, index);
            iterator.insert(child);
        }
        else if (parent is Object) {
            if (parent.children != null) {
                if(parent.children is ArrayCollection) {
                    parent.children.addItemAt(child, index);
                    if (model){
                        model.dispatchEvent(event);
                        model.itemUpdated(parent);
                    }
                    return true;
                }
                else {
                    parent.children.splice(index, 0, child);
                    if (model)
                        model.dispatchEvent(event);
                    return true;
                }
            }
        }
        return false;
    }
    // The removeChildAt method does the following:
    // If the parent parameter is null or undefined,
    // removes the child at the specified index
    // in the model.
    // If the parent parameter is an Object and has a children field,
    // removes the child at the index parameter location in the parent.
    public function removeChildAt(parent:Object, child:Object, index:int,
model:Object=null):Boolean
    {
        var event:CollectionEvent = new CollectionEvent(CollectionEvent.COLLECTION_CHANGE);
        event.kind = CollectionEventKind.REMOVE;
        event.items = [child];
        event.location = index;
        //handle top level where there is no parent
        if (!parent)
        {
            var iterator:IViewCursor = model.createCursor();
            iterator.seek(CursorBookmark.FIRST, index);
            iterator.remove();
            if (model)
                model.dispatchEvent(event);
            return true;
        }
        else if (parent is Object)
        {
            if (parent.children != undefined)
            {
                parent.children.splice(index, 1);
                if (model)
                    model.dispatchEvent(event);
                return true;
            }
        }
        return false;
    }
}
}
```

The following example uses the MyCustomTreeDataDescriptor to handle hierarchical nested ArrayCollections and objects. When you click the button, it adds a node to the tree by calling the data descriptor's `addChildAt()` method. Notice that you would not normally use the `addChildAt()` method directly. Instead, you would use the methods of a Tree or menu-based control, which in turn use the data descriptor methods to modify the data provider.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- dpcontrols\CustDataDescriptor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns="*"
    creationComplete="initCollections();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.collections.*;
            import mx.controls.treeClasses.*;
            import myComponents.*;

            /* Variables used to construct the ArrayCollection data provider
               First top-level node and its children. */
            public var nestArray1:Array = [
                {label:"item1", children: [
                    {label:"item1 child", children:     [
                        {label:"item 1 child child", data:"child data"}
                    ]}
                ]}
            ];
            /* Second top-level node and its children. */
            public var nestArray2:Array = [
                {label:"item2", children: [
                    {label:"item2 child", children: [
                        {label:"item 2 child child", data:"child data"}
                    ]}
                ]}
            ];
            /* Second top-level node and its children. */
            public var nestArray3:Array = [
                {label:"item3", children: [
                    {label:"item3 child", children: [
                        {label:"item 3 child child", data:"child data"}
                    ]}
                ]}
            ];
            /* Variable for the tree array. */
            public var treeArray:Array
            /* Variables for the three Array collections that correspond to the
               top-level nodes. */
            public var col1:ArrayCollection;
            public var col2:ArrayCollection;
            public var col3:ArrayCollection;

            /* Variable for the ArrayCollection used as the Tree data provider. */
            [Bindable]
            public var ac:ArrayCollection;

            /* Build the ac ArrayCollection from its parts. */
            public function initCollections():void{
                /* Wrap each top-level node in an ArrayCollection. */
                col1 = new ArrayCollection(nestArray1);
                col2 = new ArrayCollection(nestArray2);
                col3 = new ArrayCollection(nestArray3);
                /* Put the three top-level node
                   ArrayCollections in the treeArray. */
                treeArray = [
                    {label:"first thing", children: col1},
                    {label:"second thing", children: col2},
                    {label:"third thing", children: col3},
                ];
                /* Wrap the treeArray in an ArrayCollection. */
                ac = new ArrayCollection(treeArray);
            }
            /* Adds a child node as the first child of the selected node,
               if any. The default selectedItem is null, which causes the
               data descriptor addChild method to add it as the first child
               of the ac ArrayCollection. */
            public function clickAddChildren():void {
                var newChild:Object = new Object();
                newChild.label = "New Child";
                newChild.children = new ArrayCollection();
                tree.dataDescriptor.addChildAt(tree.selectedItem, newChild, 0, ac);
            }
        ]]>
    </fx:Script>
    <mx:Tree width="200" id="tree" dataProvider="{ac}"
        dataDescriptor="{new MyCustomTreeDataDescriptor()}"/>
    <s:Button label="Add Child" click="clickAddChildren();"/>
</s:Application>
```

XML-based data objects

The data for a tree is often retrieved from a server in the form of XML, but it can also be well-formed XML defined within the `<mx:Tree>` tag. The DefaultDataDescriptor class can handle well-formed XML data structures.

You can use an <fx:XML> or <fx:XMLList> tag to define an XML or XMLList object in MXML. Unlike the XML and XMLList classes in ActionScript, these tags let you use MXML binding expressions in the XML text to extract node contents from variable data. For example, you can bind a node's name attribute to a text input value, as in the following example:

```
<fx:XMLList id="myXMLList">
    <child name="{textInput1.text}"/>
    <child name="{textInput2.text}"/>
</fx:XMLList>
```

You can use an XML object directly as a data provider to a hierarchical data control. However, if the object changes dynamically, you should do the following:

1 Convert the XML or XMLList object to an XMLListCollection object.

2 Make all updates to the data by modifying the XMLListCollection object.

Doing this ensures that the component represents the dynamic data. The XMLListCollection class supports the use of all IList and ICollectionView interface methods, and adds many of the most commonly used XMLList class methods. For more information on using XMLListCollections, see “XMLListCollection objects” on page 937.

The following code example defines two Tree controls. The first uses an XML object directly, and the second uses an XMLListCollection object as the data source:

```
<?xml version="1.0"?>
<!-- dpcontrols\UseXMLDP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <fx:XML id="capitals">
            <root>
                <Capitals label="U.S. State Capitals">
                    <capital label="AL" value="Montgomery"/>
                    <capital label="AK" value="Juneau"/>
                    <capital label="AR" value="Little Rock"/>
                    <capital label="AZ" value="Phoenix"/>
                </Capitals>
                <Capitals label="Canadian Province Capitals">
                    <capital label="AB" value="Edmonton"/>
                    <capital label="BC" value="Victoria"/>
                    <capital label="MB" value="Winnipeg"/>
                    <capital label="NB" value="Fredericton"/>
                </Capitals>
            </root>
        </fx:XML>
        <!-- Create an XMLListCollection representing the Tree nodes.
```

```
             capitals.Capitals is an XMLList with both Capitals elements. -->
        <mx:XMLListCollection id="capitalColl" source="{capitals.Capitals}"/>
    </fx:Declarations>
    <s:Label text="These two Tree controls appear identical, although their data sources are
different."/>
    <s:HGroup>
        <!-- When you use an XML-based data provider with a tree
             you must specify the label field, even if it
             is "label". The XML object includes the root,
             so you must set showRoot="false". Remember that
             the Tree will not, by default, reflect dynamic changes
             to the XML object. -->
        <mx:Tree id="Tree1" dataProvider="{capitals}" labelField="@label"
            showRoot="false" width="300"/>

        <!-- The XMLListCollection does not include the XML root. -->
        <mx:Tree id="Tree2" dataProvider="{capitalColl}" labelField="@label"
            width="300"/>
    </s:HGroup>
</s:Application>
```

This example shows two important features of using a hierarchical data provider with a Tree control:

- ECMAScript for XML (E4X) objects must have a single root node, which might not be appropriate for displaying in the Tree. Also, trees can have multiple elements at their highest level. To prevent the tree from displaying the root node, set the `showRoot` property to `false`. (The default `showRoot` value for the Tree control is `true`.) XMLList collections, however, do not have a single root, and you typically do not need to use the `showRoot` property.
- When you use an XML, XMLList, or XMLListCollection object as the tree data provider, you must specify the `labelField` property, even if it is “label”, if the field is an XML attribute. You must do this because you must use the @ sign to signify an attribute.

XMLListCollection objects

XMLListCollection objects provide collection functionality to an XMLList object and make available some of the XML manipulation methods of the native XMLList class, such as the `attributes()`, `children()`, and `elements()` methods. For details of the supported methods, see XMLListCollectionin the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The following simple example uses an XMLListCollection object as the data provider for a List control. It uses XMLListCollection methods to dynamically add items to and remove them from the data provider and its representation in the List control. The example uses a Tree control to represent a selection of shopping items and a List collection to represent a shopping list.

Users add items to the List control by selecting an item in a Tree control (which uses a static XML object as its data provider) and clicking a button. When the user clicks the button, the event listener uses the XMListCollection `addItem()` method to add the selected XML node to the XMLListCollection. Because the data provider is a collection, the List control is updated to show the new data.

Users remove items in a similar manner, by selecting an item in the list and clicking the Remove button. The event listener uses the XMLListCollection `removeItemAt()` method to remove the item from the data provider and its representation in the List control.

```
<?xml version="1.0"?>
<!-- dpcontrols\XMLListCollectionWithList.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="550">

    <fx:Script>
        <![CDATA[
            import mx.collections.XMLListCollection;
            import mx.collections.ArrayCollection;

            /* An XML object with categorized produce. */
            [Bindable]
            public var myData:XML=
                <catalog>
                  <category name="Meat">
                      <product name="Buffalo"/>
                      <product name="T Bone Steak"/>
                      <product name="Whole Chicken"/>
                  </category>
                  <category name="Vegetables">
                      <product name="Broccoli"/>
                      <product name="Vine Ripened Tomatoes"/>
                      <product name="Yellow Peppers"/>
                  </category>
                  <category name="Fruit">
                      <product name="Bananas"/>
                      <product name="Grapes"/>
                      <product name="Strawberries"/>
                  </category>
              </catalog>;
            /* An XMLListCollection representing the data
               for the shopping List. */
           [Bindable]
           public var listDP:XMLListCollection = new XMLListCollection(new XMLList());

           /* Add the item selected in the Tree to the List XMLList data provider. */
           private function doTreeSelect():void {
               if (prodTree.selectedItem)
               listDP.addItem(prodTree.selectedItem.copy());
           }
           /* Remove the selected in the List from the XMLList data provider. */
```

```
            private function doListRemove():void {
                if (prodList.selectedItem)
                    listDP.removeItemAt(prodList.selectedIndex);
            }
        ]]>
    </fx:Script>

    <s:HGroup>
        <mx:Tree id="prodTree" dataProvider="{myData}" width="200"
            showRoot="false" labelField="@name"/>
        <s:VGroup>
            <s:Button id="treeSelect" label="Add to List"
                click="doTreeSelect()"/>
            <s:Button id="listRemove" label="Remove from List"
                click="doListRemove()"/>
        </s:VGroup>
        <s:List id="prodList" dataProvider="{listDP}" width="200"
            labelField="@name"/>
    </s:HGroup>
</s:Application>
```

Remote data in data provider components

You use Flex data access components: HTTPService, WebService, and RemoteObject, to supply data to Flex data provider components. For more information, see Accessing server-side data.

To use a remote data source to provide data, you represent the result of the remote service with the appropriate object, as follows:

- A RemoteObject component automatically returns an ArrayCollection for any data that is represented on the server as a java.util.List object, and you can use the returned object directly.
- For HTTPService and WebService results, cast the result data to a collection class if the data changes or if you use the same result in multiple places (the latter case is more efficient). As a general rule, use an ArrayCollection for serialized (list-based) objects and an XMLListCollection for XML data.

The following code snippet shows this use, casting a list returned by a web service to an Array in an ArrayCollection:

```
<mx:WebService id="employeeWS" wsdl="http://server.com/service.wsdl"
    showBusyCursor="true"
    fault="alert(event.fault.faultstring)">
    <mx:operation name="getList">
        <mx:request>
            <deptId>{dept.selectedItem.data}</deptId>
        </mx:request>
    </mx:operation>
    ...
</mx:WebService>

<mx:ArrayCollection id="ac"
    source="mx.utils.ArrayUtil.toArray(employeeWS.getList.lastResult)"/>
<mx:DataGrid dataProvider="{ac}" width="100%">
```

For more information on using data access component, see Accessing server-side data.

Handling data pages with Spark components

You can use a data collection with a remote data source. Often, you want to access the data as it arrives from the server rather than waiting for all of the remote data to load. Multiple data items in a collection can be grouped into pages. The application can then handle each page as it arrives, rather than waiting for the entire collection.

A collection that supported data pages dispatches an ItemPendingError error when a request for data item is pending. The application can then handle the error as necessary.

The MX List and MX DataGrid controls have built in support for handling ItemPendingError errors. However, the Spark DataGroup and other Spark controls, such as List, do not.

To support data paging with Spark controls, use the AsyncListView as the data provider of the control. The AsyncListView class implements the IList interface, so it can be used as the data provider of a Spark control. AsyncListView handles ItemPendingError errors thrown when items requested by a call to the `getItemAt()` method are not available.

The following example uses the AsyncListView class with a Spark List control:

```
<fx:Declarations>
    // Define an ArrayCollection to hold the data from a DataService.
    <mx:ArrayCollection id="products"/>
    <mx:DataService id="ds" destination="inventory"/>
</fx:Declarations>
// Define a Button control to populate the ArrayCollection.
<s:Button label="Get Data" click="ds.fill(products);"/>

// Wrap the ArrayCollection in an AsyncListView class to handle ItemPendingError events.
<s:List>
    <mx:AsyncListView list="{products}"/>
</s:List>
```

In this example, the AsyncListView class handles any ItemPendingError errors generated when a data item is not yet available.

The AsyncListView class defines two properties, `createPendingItemFunction` and `createFailedItemFunction`, that you can use to specify callback functions executed for an ItemPendingError error. The callback function specified by `createPendingItemFunction` creates a placeholder item in the collection when a request is pending. The callback function specified by `createFailedItemFunction` creates a placeholder item in the collection when a request fails.

The following example creates these callback functions:

```
<fx:Script>
    <![CDATA[
        import mx.collections.errors.ItemPendingError;

        private function createPendingItem(index:int, ipe:ItemPendingError):Object {
            return "[" + index + " ...]";
        }

        private function createFailedItem(index:int, info:Object):Object {
            return "[" + index + " failed]";
        }
    ]]>
</fx:Script>

<fx:Declarations>
    <mx:ArrayCollection id="products"/>
    <mx:DataService id="ds" destination="inventory"/>
</fx:Declarations>

<s:Button label="Get Data" click="ds.fill(products);"/>

<s:List>
    <mx:AsyncListView list="{products}"
        createPendingItemFunction="{createPendingItem}"
        createFailedItemFunction="{createFailedItem}"/>
</s:List>
```

Data providers and the uid property

Flex data provider controls use a unique identifier (UID) to track data items. Flex can automatically create and manage UIDs. However, there are circumstances when you must supply your own `uid` property by implementing the IUID interface, and there are circumstances when supplying your own `uid` property improves processing efficiency.

Because the Object and Array classes are dynamic, you normally do not do anything special for data objects whose items belong to these classes. However, you should consider implementing the IUID if your data object items belong to custom classes that you define.

***Note:** When Flex creates a UID for an object, such as an item in an ArrayCollection, it adds the UID as an `mx_internal_uid` property of the item. Flex creates `mx_internal_uid` properties for any objects that are dynamic and do not have bindable properties. To avoid having Flex create `mx_internal_uid` properties, the object class should do any of the following things: have at least one property with a `[Bindable]` metadata tag; implement the IUID interface; or have a `uid` property with a value.*

If Flex must consider two or more different objects to be identical, the objects must implement the IUID interface so that you can assign the same `uid` value to multiple objects. A typical case where you must implement the IUID interface is an application that uses paged collections. As the cursor moves through the collection, a particular item might be pulled down from the server and released from memory repeatedly. Every time the item is pulled into memory, a new object is created to represent the item. If you need to compare items for equality, Flex should consider all objects that represent the same item to be the same “thing.”

More common than the case where you must implement the IUID interface is the case where you can improve processing efficiency by doing so. As a general rule, you do not implement the IUID interface if the data provider elements are members of dynamic classes. Flex can automatically create a `uid` property for these classes. There is still some inefficiency, however, so you might consider implementing the IUID interface if processing efficiency is particularly important.

In all other cases, Flex uses the Dictionary mechanism to manage the `uid`, which might not be as efficient as supplying your own UID.

The IUID interface contains a single property, `uid`, which is a unique identifier for the class member, and no methods. Flex provides a UIDUtil class that uses a pseudo-random-number generator to create an identifier that conforms to the standard GUID format. Although this identifier is not guaranteed to be universally unique, it should be unique among all members of your class. To implement a class that uses the UIDUtil class, such as a Person class that has fields for a first name, last name, and ID, you can use the following pattern:

```
package {
    import mx.core.IUID;
    import mx.utils.UIDUtil;

    [Bindable]
    public class Person implements IUID {
        public var id:String;
        public var firstName:String;
        public var lastName:String;
        private var _uid:String;

        public function Person() {
            _uid = UIDUtil.createUID();
        }

        public function get uid():String {
            return _uid;
        }

        public function set uid(value:String):void {
            // Do nothing, the constructor created the uid.
        }
    }
}
```

You do not need to use the UIDUtil class in a case where the objects contain a uniquely-identifying field such as an employee ID. In this case, you can use the person's ID as the `uid` property, because the `uid` property values uniquely identify the object only in the data provider. The following example implements this approach:

```
package
{
    import mx.core.IUID;

    [Bindable]
    public class Person implements IUID {
        public var employee_id:String;
        public var firstName:String;
        public var lastName:String;

        public function get uid(): String {
            return employee_id;
        }

        public function set uid(value: String): void {
            employee_id=value;
        }
    }
}
```

***Note:** Object cloning does not manage or have a relationship with UIDs, so if you clone something that has an internal UID you must also change that internal UID. UIDs are stored on mx_internal_uid only for dynamic Objects. Instances of data classes that implement IUID store their UIDs in a uid property, so that is the property that must be changed after cloning.*

MX data-driven controls

Several MX controls take input from a data provider, an object that contains data. For example, a Tree control reads data from a data provider to define the structure of the tree and any associated data assigned to each tree node.

Several of the controls that use a data provider let you visualize complex data, and there are different ways to populate these controls by using a data provider. The following topics also provide information on data providers and controls that use data providers:

- "Data providers and collections" on page 898 contains details on data providers and how to use collections as data providers.
- "Menu-based controls" on page 985 contains information on using Menu, MenuBar, and PopUpMenuButton controls.
- "AdvancedDataGrid control" on page 1397 contains information on using the AdvancedDataGrid control, which expands on the functionality of the standard DataGrid control to add data visualization features to your Adobe Flex application.
- "OLAPDataGrid control" on page 1454 contains information on using the OLAPDataGrid control, which lets you aggregate large amounts of data for easy display and interpretation.

MX List control

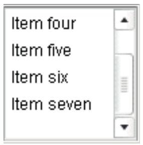
The List control displays a vertical list of items. Its functionality is very similar to that of the `SELECT` form element in HTML. It often contains a vertical scroll bar that lets users access the items in the list. An optional horizontal scroll bar lets users view items when the full width of the list items is unlikely to fit. The user can select one or more items from the list.

Flex also includes the Spark List control. When possible, Adobe recommends that you use the Spark List control. For more information, see "Spark List control" on page 525.

***Note:** The HorizontalList, TileList, DataGrid, Menu, and Tree controls are derived from the MX List control or its immediate parent, the MX ListBase class. As a result, much of the information for the List control applies to these controls.*

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The following image shows a List control:



List control sizing

If you specify `horizontalScrollPolicy="on"`, the default width of a List control does not change; it is still large enough to display the widest visible label. If you set `horizontalScrollPolicy="on"`, and specify a List control pixel width, you can use the `measureWidthOfItems()` method to ensure that the scroll bar rightmost position corresponds to the right edge of the content, as the following example shows. Notice that the additional 5 pixels ensures that the rightmost character of the text displays properly.

```
<mx:List id="li2"
    width="200" horizontalScrollPolicy="on"
    maxHorizontalScrollPosition="{li2.measureWidthOfItems() - li2.width +5}">
```

The preceding line ensures that the rightmost position of the scroll bar puts the end of the longest measured list item near the right edge of the List control. Using this technique, however, can reduce application efficiency, so you might consider using explicit sizes instead.

Lists, and all subclasses of the ListBase class, determine their sizes when a style changes or the data provider changes.

If you set a `width` property that is less than the width of the longest label and specify the `horizontalScrollPolicy="off"`, labels that exceed the control width are clipped.

Creating a List control

You use the `<mx:List>` tag to define a List control. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The List control uses a list-based data provider. For more information, see “Data providers and collections” on page 898.

You specify the data for the List control by using the `dataProvider` property of the control. However, because `dataProvider` is the List control’s default property, you do not have to specify a `dataProvider` child tag of the `<mx:List>` tag. In the simplest case for creating a static List control, you need only put `<fx:String>` tags in the control body, because Flex also automatically interprets the multiple tags as an Array of Strings, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListDataProvider.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:List>
        <mx:dataProvider>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
            <fx:String>AR</fx:String>
        </mx:dataProvider>
    </mx:List>
</s:Application>
```

Because the data in this example is inline static data, it is not necessary to explicitly wrap it in an ArrayList or ArrayCollection object. However, when working with data that could change, it is always best to specify a list or collection explicitly; for more information, see “Data providers and collections” on page 898.

The index of items in the List control is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items. The value of the item is its label text.

You typically use events to handle user interaction with a List control. The following example code adds a handler for a `change` event to the List control. Flex broadcasts a `mx.ListEvent.CHANGE` event when the value of the control changes due to user interaction.

```
<?xml version="1.0"?>
<!-- dpcontrols/ListChangeEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            public function changeEvt(event:Event):void {
                forChange.text=event.currentTarget.selectedItem.label + " " +
                event.currentTarget.selectedIndex;
            }
        ]]>
    </fx:Script>
    <mx:List width="35" change="changeEvt(event)">
        <fx:Object label="AL" data="Montgomery"/>
        <fx:Object label="AK" data="Juneau"/>
        <fx:Object label="AR" data="Little Rock"/>
    </mx:List>
    <mx:TextArea id="forChange" width="150"/>
</s:Application>
```

In this example, you use two properties of the List control, `selectedItem` and `selectedIndex`, in the event handler. Every `change` event updates the TextArea control with the label of the selected item and the item's index in the control.

The `target` property of the object passed to the event handler contains a reference to the List control. You can reference any control property by using the event's `currentTarget` property. The `currentTarget.selectedItem` field contains a copy of the selected item. If you populate the List control with an Array of Strings, the `currentTarget.selectedItem` field contains a String. If you populate it with an Array of Objects, the `currentTarget.selectedItem` field contains the Object that corresponds to the selected item, so, in this case, `currentTarget.selectedItem.label` refers to the selected item's label field.

Using a label function

You can pass a label function to the List control to provide logic that determines the text that appears in the control. The label function must have the following signature:

```
labelFunction(item:Object):String
```

The *`item`* parameter passed in by the Label control contains the list item object. The function must return the string to display in the List control.

***Note:** Most subclasses of ListBase also take a `labelFunction` property with the signature described above. For the DataGrid and DataGridColumn controls, the method signature is `labelFunction(item:Object, dataField:DataGridColumn):String`, where `item` contains the DataGrid item object, and `dataField` specifies the DataGrid column.*

The following example uses a function to combine the values of the `label` and `data` fields for each item for display in the List control:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListLabelFunction.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >

    <fx:Script>
        <![CDATA[
            public function myLabelFunc(item:Object):String {
                return item.data + ", " + item.label;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ArrayList id="myDP">
            <mx:source>
                <fx:Object label="AL" data="Montgomery"/>
                <fx:Object label="AK" data="Juneau"/>
                <fx:Object label="AR" data="Little Rock"/>
            </mx:source>
        </mx:ArrayList>
    </fx:Declarations>
    <mx:List dataProvider="{myDP}" labelFunction="myLabelFunc"/>
</s:Application>
```

***Note:** This example uses an ArrayList object as the data provider. You should always use a list or collection as the data provider when the data can change at run time. For more information, see "Data providers and collections" on page 898.*

Displaying DataTips

DataTips are similar to ToolTips, but display text when the mouse pointer hovers over a row in a List control. Text in a List control that is longer than the control width is clipped on the right side (or requires scrolling, if the control has scroll bars). DataTips can solve that problem by displaying all of the text, including the clipped text, when the mouse pointer hovers over a cell. If you enable data tips, they only appear for fields where the data is clipped. To display DataTips, set the `showDataTips` property of a List control to `true`.

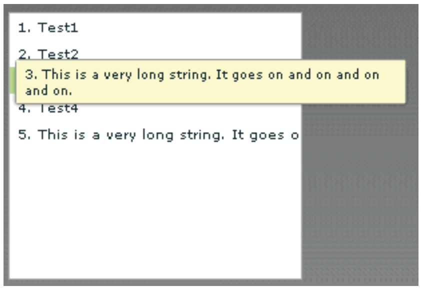
***Note:** To use DataTips with a DataGrid control, you must set the `showDataTips` property on the individual DataGridColumns of the DataGrid.*

The default behavior of the `showDataTips` property is to display the label text. However, you can use the `dataTipField` and `dataTipFunction` properties to determine what is displayed in the DataTip. The `dataTipField` property behaves like the `labelField` property; it specifies the name of the field in the data provider to use as the DataTip for cells in the column. The `dataTipFunction` property behaves like the `labelFunction` property; it specifies the DataTip string to display for list items.

The following example sets the `showDataTips` property for a List control:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200" showDataTips="true"/>
```

This example creates the following List control:



1. Test1
2. Test2
3. This is a very long string. It goes on and on and on and on.
4. Test4
5. This is a very long string. It goes o

Displaying ScrollTips

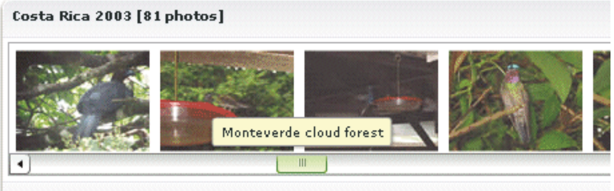
You use ScrollTips to give users context about where they are in a list as they scroll through the list. The tips appear only when you scroll; they don't appear if you only hover the mouse over the scroll bar. ScrollTips are useful when live scrolling is disabled (the `liveScrolling` property is false) so scrolling does not occur until you release the scroll thumb. The default value of the `showScrollTips` property is `false`.

The default behavior of the `showScrollTips` property is to display the index number of the top visible item. You can use the `scrollTipFunction` property to determine what is displayed in the ScrollTip. The `scrollTipFunction` property behaves like the `labelFunction` property; it specifies the ScrollTip string to display for list items. You should avoid going to the server to fill in a ScrollTip.

The following example sets the `showScrollTips` and `scrollTipFunction` properties of a HorizontalList control, which shares many of the same properties and methods as the standard List control; for more information about the HorizontalList control, see "MX HorizontalList control" on page 951. The `scrollTipFunction` property specifies a function that gets the value of the `description` property of the current list item.

```
<mx:HorizontalList id="list" dataProvider="{album.photo}" width="100%"
    itemRenderer="Thumbnail" columnWidth="108" height="100"
    selectionColor="#FFCC00" liveScrolling="false" showScrollTips="true"
    scrollTipFunction="scrollTipFunc"
    change="currentPhoto=album.photo[list.selectedIndex]"/>
```

This code produces the following HorizontalList control:



Vertically aligning text in List control rows

You can use the `verticalAlign` style to vertically align text at the top, middle, or bottom of a List row. The default value is `top`. You can also specify a value of `middle` or `bottom`.

The following example sets the `verticalAlign` property for a List control to `bottom`:

```
<mx:List id="myList" dataProvider="{myDP}"
    width="220" height="200"
    verticalAlign="bottom"/>
```

Setting variable row height and wrapping List text

You can use the `variableRowHeight` property to make the height of List control rows variable based on their content. The default value is `false`. If you set the `variableRowHeight` property to `true`, the `rowHeight` property is ignored and the `rowCount` property is read-only.

The following example sets the `variableRowHeight` property for a List control to `true`:

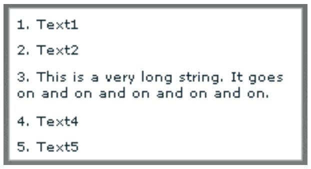
```
<mx:List id="myList" dataProvider="{myDP}"
    width="220" height="200"
    variableRowHeight="true"/>
```

You can use the `wordWrap` property in combination with the `variableRowHeight` property to wrap text to multiple lines when it exceeds the width of a List row.

The following example sets the `wordWrap` and `variableRowHeight` properties to `true`:

```
<mx:List id="myList" dataProvider="{myDP}"
    width="220" height="200"
    variableRowHeight="true" wordWrap="true"/>
```

This code produces the following List control:



1. Text1
2. Text2
3. This is a very long string. It goes on and on and on and on and on.
4. Text4
5. Text5

Alternating row colors in a List control

You can use the `alternatingItemColors` style property to specify an Array that defines the color of each row in the List control. The Array must contain two or more colors. After using all the entries in the Array, the List control repeats the color scheme.

The following example defines an Array with two entries, #66FFFF for light blue and #33CCCC for a blue-gray. Therefore, the rows of the List control alternate between these two colors. If you specify a three-color array, the rows alternate among the three colors, and so on.

```
<mx:List alternatingItemColors="[#66FFFF, #33CCCC]".../>
```

Using a custom item renderer

An item renderer is the object that displays a List control’s data items. The simplest way to use a custom item renderer is to specify an MXML component as the value of the `itemRenderer` property. When you use an MXML component as a item renderer, it can contain multiple levels of containers and controls. You can also use an ActionScript class as a custom item renderer. For detailed information on custom item renderers, see “MX item renderers and item editors” on page 1006

The following example sets the `itemRenderer` property to an MXML component named FancyCellRenderer. It also sets the `variableRowHeight` property to `true` because the MXML component exceeds the default row height:

```
<mx:List id="myList1" dataProvider="{myDP}"
    width="220" height="200"
    itemRenderer="FancyItemRenderer"
    variableRowHeight="true"/>
```

Specifying an icon to the List control

You can specify an icon to display with each List item, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListIcon.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            [Embed(source="assets/radioIcon.jpg")]
            public var iconSymbol1:Class;
            [Embed(source="assets/topIcon.jpg")]
            public var iconSymbol2:Class;
        ]]>
    </fx:Script>
   <mx:List iconField="myIcon">
      <mx:dataProvider>
         <fx:Array>
            <fx:Object label="AL" data="Montgomery" myIcon="iconSymbol1"/>
            <fx:Object label="AK" data="Juneau" myIcon="iconSymbol2"/>
            <fx:Object label="AR" data="Little Rock" myIcon="iconSymbol1"/>
         </fx:Array>
      </mx:dataProvider>
   </mx:List>
</s:Application>
```

In this example, you use the `iconField` property to specify the field of each item containing the icon. You use the `Embed` metadata to import the icons, and then reference them in the List control definition.

You can also use the `iconFunction` property to specify a function that determines the icon, similar to the way that you can use the `labelFunction` property to specify a function that determines the label text. The icon function must have the following signature:

```
iconFunction(item:Object):Class
```

The *`item`* parameter passed in by the List control contains the list item object. The function must return the icon class to display in the List control.

The following example shows a List control that uses the `iconFunction` property to determine the icon to display for each item in the list:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListIconFunction.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            // Embed icons.
            [Embed(source="assets/radioIcon.jpg")]
            public var pavementSymbol:Class;
            [Embed(source="assets/topIcon.jpg")]
            public var normalSymbol:Class;

            // Define data provider.
            private var myDP: Array;
            private function initList():void {
                myDP = [
                    {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
                    {Artist:'Pavarotti', Album:'Twilight', Price:11.99},
                    {Artist:'Other', Album:'Other', Price:5.99}];

                list1.dataProvider = myDP;
            }

            // Determine icon based on artist. Pavement gets a special icon.
            private function myiconfunction(item:Object):Class {
                var type:String = item.Artist;
                if (type == "Pavement") {
                    return pavementSymbol;
                }
                return normalSymbol;
            }
        ]]>
    </fx:Script>

    <mx:VBox >
        <mx:List id="list1"
            initialize="initList();"
            labelField="Artist"
            iconFunction="myiconfunction" />
    </mx:VBox>
</s:Application>
```

List control user interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items. (You must set the `allowMultipleSelection` property to `true` to allow multiple selection.)

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the List control broadcasts this event when the mouse button is released.

If you set the `allowDragSelection` property to `true`, the control scrolls up or down when the user presses the mouse button over the one or more rows, holds the mouse button down, drags the mouse outside the control, and then moves the mouse up and down.

A List control shows the number of records that fit in the display. Paging down through the data displayed by a 10-line List control shows records 0-10, 9-18, 18-27, and so on, with one line overlapping from one page to the next.

The List control has the following keyboard navigation features:

Key	Action
Up Arrow	Moves selection up one item.
Down Arrow	Moves selection down one item.
Page Up	Moves selection up one page.
Page Down	Moves selection down one page.
Home	Moves selection to the top of the list.
End	Moves selection to the bottom of the list.
Alphanumeric keys	Jumps to the next item with a label that begins with the character typed.
Control	Toggle key. Allows for multiple (noncontiguous) selection and deselection. Works with key presses, click selection, and drag selection.
Shift	Contiguous selection key. Allows for contiguous selections. Works with key presses, click selection, and drag selection.

MX HorizontalList control

The HorizontalList control displays a horizontal list of items. The HorizontalList control is particularly useful in combination with a custom item renderer for displaying a list of images and other data. For more information about custom item renderers, see "MX item renderers and item editors" on page 1006.

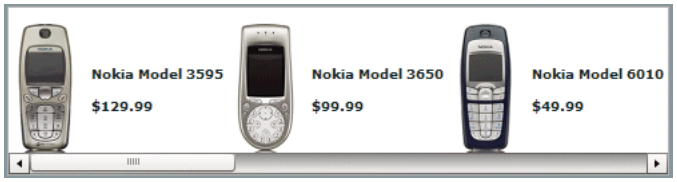
For complete reference information, see HorizontalList in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About HorizontaList controls

The contents of a HorizontalList control can look very similar to the contents of an HBox container in which a Repeater object repeats components. However, performance of a HorizontalList control can be better than the combination of an HBox container and a Repeater object because the HorizontalList control only instantiates the objects that fit in its display area. Scrolling in a HorizontalList can be slower than it is when using a Repeater object. For more information about the Repeater object, see "Dynamically repeating controls and containers" on page 867.

The HorizontalList control always displays items from left to right. The control usually contains a horizontal scroll bar, which lets users access all items in the list. An optional vertical scroll bar lets users view items when the full height of the list items is unlikely to fit. The user can select one or more items from the list, depending on the value of the `allowMultipleSelection` property.

The following image shows a HorizontalList control:



For complete reference information, see HorizontalList in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a HorizontalList control

You use the `<mx:HorizontalList>` tag to define a HorizontalList control. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The HorizontalList control shares many properties and methods with the List control; see “MX List control” on page 943 for information on how to use several of these shared properties. The HorizontalList control uses a list-based data provider. For more information, see “Data providers and collections” on page 898.

You specify the data for a HorizontalList control by using the `dataProvider` property of the `<mx:HorizontalList>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/HListDataProvider.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="450">

    <fx:Script>
        <![CDATA[
            import mx.collections.*;
            import mx.controls.Image;
            private var catalog:ArrayList;
            private static var cat:Array = [
                "../assets/Nokia_6820.gif", "../assets/Nokia_3595.gif",
                "../assets/Nokia_3650.gif", "../assets/Nokia_6010.gif"
            ];
            /* Initialize the HorizontalList control by setting its dataProvider
                property to an ArrayList containing the items parameter. */
            private function initCatalog(items:Array):void {
                catalog = new ArrayList(items);
                myList.dataProvider = catalog;
            }
        ]]>
    </fx:Script>
    <!-- A four-column HorizontalList. The itemRenderer is a Flex Image control.
       When the control is created, pass the cat array to the initialization routine. -->
    <mx:HorizontalList id="myList"
        columnWidth="100"
        rowHeight="100"
        columnCount="4"
        itemRenderer="mx.controls.Image"
        creationComplete="initCatalog(cat);"/>
</s:Application>
```

In this example, you use the `creationComplete` event to populate the data provider with an ArrayList of image files, and the `itemRenderer` property to specify the Image control as the item renderer. (Note that you use the full package name of the control in the assignment because the code does not import the mx.controls package.) The HorizontalList control then displays the four images specified by the data provider.

HorizontalList control user interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items.

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the HorizontalList control broadcasts this event when the mouse button is released. When the user drags the mouse over the items and then outside the control, the control scrolls up or down.

A HorizontalList control shows the number of records that fit in the display. Paging through a four list shows records 0-4, 5-8, and so on, with no overlap from one page to the next.

Keyboard navigation

The HorizontalList control has the following keyboard navigation features:

Key	Action
Page Up	Moves selection to the left one page.
Left Arrow	Moves selection to the left one item.
Down Arrow	Moves selection right one item.
Page Down	Moves selection to the right one page.
Home	Moves selection to the beginning of the list.
End	Moves selection to the end of the list.
Control	Toggle key. Allows for multiple (noncontiguous) selection and deselection when the `allowMultipleSelection` property is set to `true`. Works with key presses, click selection, and drag selection.
Shift	Contiguous selection key. Allows for contiguous selections when `allowMultipleSelection` is set to `true`. Works with key presses, click selection, and drag selection.

MX TileList control

The TileList control displays a tiled list of items. The items are tiled in vertical columns or horizontal rows. The TileList control is particularly useful in combination with a custom item renderer for displaying a list of images and other data. The default item renderer for the TileList control is TileListItemRenderer, which, by default, displays text of the data provider's label field and any icon. For more information about custom item renderers, see "MX item renderers and item editors" on page 1006.

Flex also includes the Spark Tile layout. When possible, Adobe recommends that you use the Spark Tile layout with the Spark List control. For more information, see "Spark List control" on page 525.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

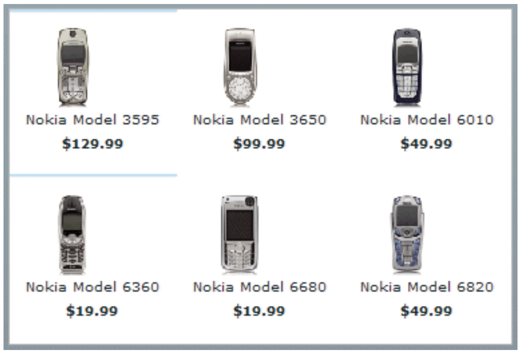
About the TileList control

The contents of a TileList control can look very similar to the contents of a Tile container in which a Repeater object repeats components. However, performance of a TileList control can be better than the combination of a Tile container and a Repeater object because the TileList control only instantiates the objects that fit in its display area. Scrolling in a TileList can be slower than it is when using a Repeater object. For more information about the Repeater object, see "Dynamically repeating controls and containers" on page 867.

The TileList control displays a number of items laid out in equally sized tiles. It often contains a scroll bar on one of its axes to access all items in the list depending on the direction orientation of the list. The user can select one or more items from the list depending on the value of the `allowMultipleSelection` property.

The TileList control lays out its children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The direction property determines the primary direction of the layout. The valid values for the direction property are and horizontal (default) and vertical for a layout. In a horizontal layout the tiles are filled in row by row with each row filling the available space in the control. If there are more tiles than fit in the display area, the horizontal control has a vertical scroll. In a vertical layout, the tiles are filled in column by column in the available vertical space, and the control may have a horizontal scroll bar.

The following image shows a TileList control:



Creating a TileList control

You use the <mx:TileList> tag to define a TileList control. Specify an id value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The TileList control shares many properties and methods with the List control; see “MX List control” on page 943 for information on how to use several of these shared properties. The TileList control uses a list-based data provider. For more information, see “Data providers and collections” on page 898.

You specify the data for a TileList control by using the dataProvider property of the <mx:TileList> tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/TileListDataProvider.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();" >

    <fx:Script>
        <![CDATA[
            import mx.controls.Button;
            import mx.collections.*;

            private var listArray:Array=[
                {label: "item0", data: 0},{label: "item1", data: 1},
                {label: "item2", data: 2},{label: "item3", data: 3},
                {label: "item4", data: 4},{label: "item5", data: 5},
                {label: "item6", data: 6},{label: "item7", data: 7},
                {label: "item8", data: 8}];

            [Bindable]
            public var TileListdp:ArrayList;

            private function initData():void {
                TileListdp = new ArrayList(listArray);
            }
        ]]>
    </fx:Script>
   <mx:TileList dataProvider="{TileListdp}"
      itemRenderer="mx.controls.Button"/>
</s:Application>
```

In this example, you populate the data provider with an ArrayList that contains an Array of strings defining labels and data values. You then use the `itemRenderer` property to specify a Button control as the item renderer. The Button controls display the data provider label values. The TileList control displays nine Button controls with the specified labels.

TileList control user interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items.

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the TileList control broadcasts this event when the mouse button is released. When the user drags the mouse over the items and then outside the control, the control scrolls up or down.

Keyboard navigation

The TileList control has the following keyboard navigation features:

Key	Action
Up Arrow	Moves selection up one item. If the control direction is `vertical`, and the current item is at the top of a column, moves to the last item in the previous column; motion stops at the first item in the first column.
Down Arrow	Moves selection down one item. If the control direction is `vertical`, and the current item is at the bottom of a column, moves to the first item in the next column; motion stops at the last item in the last column.
Right Arrow	Moves selection to the right one item. If the control direction is `horizontal`, and the current item is at the end of a row, moves to the first item in the next row; motion stops at the last item in the last column.
Left Arrow	Moves selection to the left one item. If the control direction is `horizontal`, and the current item is at the beginning of a row, moves to the last item in the previous row; motion stops at the first item in the first row.
Page Up	Moves selection up one page. For a single-page control, moves the selection to the beginning of the list.
Page Down	Moves selection down one page. For a single-page control, moves the selection to the end of the list.
Home	Moves selection to the beginning of the list.
End	Moves selection to the end of the list.
Control	Toggle key. Allows for multiple (noncontiguous) selection and deselection when `allowMultipleSelection` is set to `true`. Works with key presses, click selection, and drag selection.
Shift	Contiguous selection key. Allows for contiguous selections when `allowMultipleSelection` is set to `true`. Works with key presses, click selection, and drag selection.

MX ComboBox control

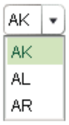
The ComboBox control is a drop-down list from which the user can select a single value. Its functionality is very similar to that of the SELECT form element in HTML.

Flex also includes the Spark ComboBox control. When possible, Adobe recommends that you use the Spark ComboBox control. For more information, see “Spark ComboBox control” on page 536.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About the ComboBox control

The following image shows a ComboBox control:



In its editable state, the user can type text directly into the top of the list, or select one of the preset values from the list. In its noneditable state, as the user types a letter, the drop-down list opens and scrolls to the value that most closely matches the one being entered; matching is only performed on the first letter that the user types.

If the drop-down list hits the lower boundary of the application, it opens upward. If a list item is too long to fit in the horizontal display area, it is truncated to fit. If there are too many items to display in the drop-down list, a vertical scroll bar appears.

Creating a ComboBox control

You use the `<mx:ComboBox>` tag to define a ComboBox control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The ComboBox control uses a list-based data provider. For more information, see "Data providers and collections" on page 898.

You specify the data for the ComboBox control by using the `dataProvider` property of the `<mx:ComboBox>` tag. The data provider should be a list or collection; the standard list implementation is the ArrayList class. The standard collection implementations are the ArrayCollection and XMLListCollection classes, for working with Array-based and XML-based data, respectively. You can also use a raw data object, such as an Array object, as a data provider; however, it is always better to specify a list or collection explicitly for data that could change. For more information on data providers and collections see "Data providers and collections" on page 898.

In a simple case for creating a ComboBox control, you specify the property by using an `dataProvider` child tag, and use an `<mx:ArrayList>` tag to define the entries as an ArrayList whose source is an Array of Strings, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

   <mx:ComboBox>
      <mx:ArrayList>
         <fx:String>AK</fx:String>
         <fx:String>AL</fx:String>
         <fx:String>AR</fx:String>
      </mx:ArrayList>
   </mx:ComboBox>
</s:Application>
```

This example shows how you can take advantages of MXML defaults. You do not have to use an `dataProvider` tag, because `dataProvider` is the default property of the ComboBox control. Similarly, you do not have to use an `<mx:source>` tag inside the `<mx:ArrayList>` tag because `source` is the default property of the ArrayList class. Finally, you do not have to specify an `<fx:Array>` tag for the source array.

The data provider can also contain objects with multiple fields, as in the following example:

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxMultiple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

   <mx:ComboBox>
      <mx:ArrayList>
         <fx:Object label="AL" data="Montgomery"/>
         <fx:Object label="AK" data="Juneau"/>
         <fx:Object label="AR" data="Little Rock"/>
      </mx:ArrayList>
   </mx:ComboBox>
</s:Application>
```

If the data source is an array of strings, as in the first example, the ComboBox displays strings as the items in the drop-down list. If the data source consists of objects, the ComboBox, by default, uses the contents of the label field. You can, however, override this behavior, as described in "Specifying ComboBox labels" on page 959.

The index of items in the ComboBox control is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items. The value of the item is its label text.

Using events with ComboBox controls

You typically use events to handle user interaction with a ComboBox control.

The ComboBox control broadcasts a `change` event (flash.events.Event class with a `type` property value of `flash.events.Event.CHANGE`) when the value of the control's `selectedIndex` or `selectedItem` property changes due to the following user actions:

- If the user closes the drop-down list by using a mouse click, Enter key, or Control+Up key, and the selected item is different from the previously selected item.
- If the drop-down list is currently closed, and the user presses the Up, Down, Page Up, or Page Down key to select a new item.
- If the ComboBox control is editable, and the user types into the control, Flex broadcasts a `change` event each time the text field of the control changes.

The ComboBox control broadcasts an mx.events.DropdownEvent with a type `mx.events.DropdownEvent.OPEN` (open`)` and `mx.events.DropdownEvent.CLOSE` (close) when the ComboBox control opens and closes. For detailed information on these and other ComboBox events, see ComboBox in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The following example displays information from ComboBox events:

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            import mx.events.DropdownEvent;
            // Display the type of event for open and close events.
            private function dropEvt(event:DropdownEvent):void {
                forChange.text+=event.type + "\n";
            }
            // Display a selected item's label field and index for change events.
            private function changeEvt(event:Event):void {
                forChange.text+=event.currentTarget.selectedItem.label + " " +
                event.currentTarget.selectedIndex + "\n";
            }
        ]]>
    </fx:Script>
    <mx:ComboBox open="dropEvt(event)" close="dropEvt(event)"
        change="changeEvt(event)" >
        <mx:ArrayList>
            <fx:Object label="AL" data="Montgomery"/>
            <fx:Object label="AK" data="Juneau"/>
            <fx:Object label="AR" data="Little Rock"/>
        </mx:ArrayList>
    </mx:ComboBox>
    <mx:TextArea id="forChange" width="150" height="150"/>
</s:Application>
```

If you populate the ComboBox control with an Array of Strings, the `currentTarget.selectedItem` field contains a String. If you populate it with an Array of Objects, the `currentTarget.selectedItem` field contains the Object that corresponds to the selected item, so, in this case, `currentTarget.selectedItem.label` refers to the selected item object's label field.

In this example, you use two properties of the ComboBox control, `selectedItem` and `selectedIndex`, in the event handlers. Every `change` event updates the TextArea control with the label of the selected item and the item's index in the control, and every `open` or `close` event appends the event type.

Specifying ComboBox labels

If the ComboBox data source is an array of strings, the control displays the string for each item. If the data source is contains Objects, by default, the ComboBox control expects each object to contain a property named `label` that defines the text that appears in the ComboBox control for the item. If each Object does not contain a `label` property, you can use the `labelField` property of the ComboBox control to specify the property name, as the following example shows:

```
    <mx:ComboBox open="dropEvt(event)" close="dropEvt(event)"
            change="changeEvt(event)" labelField="state">
        <mx:ArrayList>
            <fx:Object state="AL" capital="Montgomery"/>
            <fx:Object state="AK" capital="Juneau"/>
            <fx:Object state="AR" capital="Little Rock"/>
        </mx:ArrayList>
    </mx:ComboBox>
```

To make the event handler in the section “Using events with ComboBox controls” on page 958display the state ID and capital, you would modify the `change` event handler to use a property named `state`, as the following example shows:

```
        private function changeEvt(event) {
            forChange.text=event.currentTarget.selectedItem.state + " " +
            event.currentTarget.selectedItem.capital + " " +
            event.currentTarget.selectedIndex;
        }
```

You can also specify the ComboBox labels by using a label function, as described in “Using a label function” on page 945.

Populating a ComboBox control by using variables and models

Flex lets you populate the data provider of a ComboBox control from an ActionScript variable definition or from a Flex data model. Each element of the data provider must contain a string label, and can contain one or more fields with additional data. The following example populates two ComboBox controls; one from an ArrayList variable that it populates directly from an Array, the other from an ArrayList that it populates from an array of items in an `<fx:Model>` tag.

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxVariables.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.collections.*
            private var COLOR_ARRAY:Array=
                [{label:"Red", data:"#FF0000"},
                {label:"Green", data:"#00FF00"},
                {label:"Blue", data:"#0000FF"}];
            // Declare an ArrayList variable for the colors.
            // Make it Bindable so it can be used in bind
            // expressions ({colorAL}).
            [Bindable]
            public var colorAL:ArrayList;
            // Initialize colorAL ArrayList variable from the Array.
            // Use an initialize event handler to initialize data variables
            // that do not rely on components, so that the initial values are
            // available when the controls that use them are constructed.
            //See the mx:ArrayList tag, below, for a second way to
            //initialize an ArrayList.
            private function initData():void {
                colorAL=new ArrayList(COLOR_ARRAY);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- This example shows two different ways to
            structure a Model. -->
        <fx:Model id="myDP">
            <obj>
                <item label="AL" data="Montgomery"/>
                <item>
                    <label>AK</label>
                    <data>Juneau</data>
                </item>
                <item>
                    <label>AR</label>
                    <data>Little Rock</data>
                </item>
            </obj>
        </fx:Model>

       <!-- Create a stateAL ArrayList that uses as its source an Array of
             the item elements from the myDP model.
             This technique and the declaration and initialization code used for
             the colorAL variable are alternative methods of creating and
             initializing the ArrayList. -->
       <mx:ArrayList id="stateAL" source="{myDP.item}"/>
    </fx:Declarations>
   <mx:ComboBox dataProvider="{colorAL}"/>
   <mx:ComboBox dataProvider="{stateAL}"/>
</s:Application>
```

This example uses a simple model. However, you can populate the model from an external data source or define a custom data model class in ActionScript. For more information on using data models, see "Storing data" on page 889.

You can use remote data providers to supply data to your ComboBox control. For example, when a web service operation returns an Array of strings, you can use the following format to display each string as a row of a ComboBox control:

```
<mx:ArrayCollection id="resultAC"
    source="mx.utils.ArrayUtil.toArray(service.operation.lastResult);"
<mx:ComboBox dataProvider="{resultAC}" />
```

In general, you should use an ArrayCollection for web service operations. An ArrayList is not usually compatible with the results. For more information on using remote data providers, see "Remote data in data provider components" on page 939.

ComboBox control user interaction

A ComboBox control can be noneditable or editable, as specified by the Boolean `editable` property. In a noneditable ComboBox control, a user can make a single selection from a drop-down list. In an editable ComboBox control, the portion button of the control is a text field that the user can enter text directly into or can populate by selecting an item from the drop-down list. When the user makes a selection in the ComboBox control list, the label of the selection is copied to the text field at the top of the ComboBox control.

When a ComboBox control (and not the drop-down box) has focus and is editable, all keystrokes go to the text field and are handled according to the rules of the TextInput control (see “MX TextInput control” on page 825), with the exception of the Control+Down key combination, which opens the drop-down list. When the drop-down list is open, you can use the Up and Down keys to navigate in the list and the Enter key to select an item from the list.

When a ComboBox control has focus and is noneditable, alphanumeric keystrokes move the selection up and down the data provider to the next item with the same first character and display the label in the text field. If the drop-down list is open, the visible selection moves to the selected item.

You can also use the following keys to control a noneditable ComboBox control when the drop-down list is not open:

Key	Description
Control+Down	Opens the drop-down list and gives it focus.
Down	Moves the selection down one item.
End	Moves the selection to the bottom of the collection.
Home	Moves the selection to the top of the collection.
Page Down	Displays the item that would be at the end bottom of the drop-down list. If the current selection is a multiple of the `rowCount` value, displays the item that `rowCount` -1 down the list, or the last item. If the current selection is the last item in the data provider, does nothing.
Page Up	Displays the item that would be at the top of the drop-down. If the current selection is a multiple of the `rowCount` value, displays the item that `rowCount` -1 up the list, or the first item. If the current selection is the first item in the data provider, does nothing.
Up	Moves the selection up one item.

When the drop-down list of a noneditable ComboBox control has focus, alphanumeric keystrokes move the selection up and down the drop-down list to the next item with the same first character. You can also use the following keys to control a drop-down list when it is open:

Key	Description
Control+Up	Closes the drop-down list and returns focus to the ComboBox control.
Down	Moves the selection down one item.
End	Moves the selection to the bottom of the collection.
Enter	Closes the drop-down list and returns focus to the ComboBox control.
Escape	Closes the drop-down list and returns focus to the ComboBox control.
Home	Moves the selection to the top of the collection.
Page Down	Moves to the bottom of the visible list. If the current selection is at the bottom of the list, moves the current selection to the top of the displayed list and displays the next `rowCount`-1 items, if any. If there current selection is the last item in the data provider, does nothing.
Page Up	Moves to the top of the visible list. If the current selection is at the top of the list, moves the current selection to the bottom of the displayed list and displays the previous `rowCount`-1 items, if any. If the current selection is the first item in the data provider, does nothing.
Shift+Tab	Closes the drop-down list and moves the focus to the previous object in the DisplayList.
Tab	Closes the drop-down list and moves the focus to the next object in the DisplayList.
Up	Moves the selection up one item.

MX DataGrid control

The DataGrid control is a list that can display more than one column of data. It is a formatted table of data that lets you set editable table cells, and is the foundation of many data-driven applications.

For information on the following topics, which are often important for creating advanced data grid controls, see:

- How to format the information in each DataGrid cell and control how users enter data in the cells; see “MX item renderers and item editors” on page 1006.
- How to drag objects to and from the data grid; see “Drag and drop” on page 1893.

Flex also includes the Spark DataGrid control. When possible, Adobe recommends that you use the Spark DataGrid control. For more information, see “Spark DataGrid and Grid controls” on page 545.

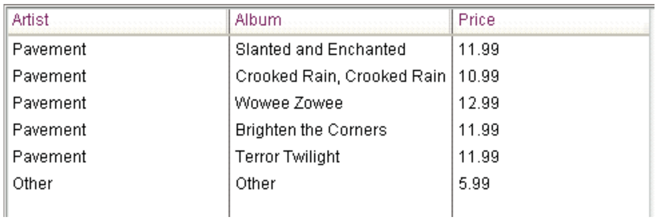
For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About the MX DataGrid control

The DataGrid control provides the following features:

- Resizable, sortable, and customizable column layouts, including hidable columns
- Optional customizable column and row headers, including optionally wrapping header text
- Columns that the user can resize and reorder at run time
- Selection events
- Ability to use a custom item renderer for any column
- Support for paging through data
- Locked rows and columns that do not scroll

The following image shows a DataGrid control:



Artist	Album	Price
Pavement	Slanted and Enchanted	11.99
Pavement	Crooked Rain, Crooked Rain	10.99
Pavement	Wowee Zowee	12.99
Pavement	Brighten the Corners	11.99
Pavement	Terror Twilight	11.99
Other	Other	5.99

Rows are responsible for rendering items. Each row is laid out vertically below the previous one. Columns are responsible for maintaining the state of each visual column; columns control width, color, and size.

Creating an MX DataGrid control

You use the `<mx:DataGrid>` tag to define a DataGrid control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The DataGrid control uses a list-based data provider. For more information, see “Data providers and collections” on page 898.

You specify the data for the DataGrid control by using the `dataProvider` property. You can specify data in several different ways. In the simplest case for creating a DataGrid control, you use the `dataProvider` property subtag with `<mx:ArrayList>`, and `<fx:Object>` tags to define the entries as an ArrayList of Objects. Each Object defines a row of the DataGrid control, and properties of the Object define the column entries for the row, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

   <mx:DataGrid>
      <mx:ArrayList>
         <fx:Object>
            <fx:Artist>Pavement</fx:Artist>
            <fx:Price>11.99</fx:Price>
            <fx:Album>Slanted and Enchanted</fx:Album>
         </fx:Object>
         <fx:Object>
            <fx:Artist>Pavement</fx:Artist>
            <fx:Album>Brighten the Corners</fx:Album>
            <fx:Price>11.99</fx:Price>
         </fx:Object>
      </mx:ArrayList>
   </mx:DataGrid>
</s:Application>
```

This example shows how you can take advantages of MXML defaults. You do not have to use an `dataProvider` tag, because `dataProvider` is the default property of the DataGrid control. Similarly, you do not have to use an `<mx:source>` tag inside the `<mx:ArrayList>` tag because `source` is the default property of the ArrayList class. Finally, you do not have to specify an `<fx:Array>` tag for the source array.

You can also define the objects by using properties directly in the Object tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSimpleAttributes.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

   <mx:DataGrid>
      <mx:ArrayList>
         <fx:Object Artist="Pavement"
            Album="Slanted and Enchanted" Price="11.99" />
         <fx:Object Artist="Pavement"
            Album="Brighten the Corners" Price="11.99" />
      </mx:ArrayList>
   </mx:DataGrid>
</s:Application>
```

The column names displayed in the DataGrid control are the property names of the Array Objects. By default, the columns are in alphabetical order by the property names. Different Objects can define their properties in differing orders. If an Array Object omits a property, the DataGrid control displays an empty cell in that row.

Specifying columns

Each column in a DataGrid control is represented by a DataGridColumn object. You use the `columns` property of the DataGrid control and the `<mx:DataGridColumn>` tag to select the DataGrid columns, specify the order in which to display them, and set additional properties. You can also use the DataGridColumn class `visible` property to hide and redisplay columns, as described in the next section.

For complete reference information for the `<mx:DataGridColumn>` tag, see DataGridColumn in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

You specify an Array element to the `<mx:columns>` child tag of the `<mx:DataGrid>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSpecifyColumns.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

   <mx:DataGrid>
      <mx:ArrayList>
         <fx:Object Artist="Pavement" Price="11.99"
            Album="Slanted and Enchanted" />
         <fx:Object Artist="Pavement"
            Album="Brighten the Corners" Price="11.99" />
      </mx:ArrayList>
      <mx:columns>
         <mx:DataGridColumn dataField="Album" />
         <mx:DataGridColumn dataField="Price" />
      </mx:columns>
   </mx:DataGrid>
</s:Application>
```

In this example, you only display the Album and Price columns in the DataGrid control. You can reorder the columns as well, as the following example shows:

```
<mx:columns>
    <mx:DataGridColumn dataField="Price" />
    <mx:DataGridColumn dataField="Album" />
</mx:columns>
```

In this example, you specify that the Price column is the first column in the DataGrid control, and that the Album column is the second.

You can also use the `<mx:DataGridColumn>` tag to set other options. The following example uses the `headerText` property to set the name of the column to a value different than the default name of Album, and uses the `width` property to set the album name column wide enough to display the full album names:

```
<mx:columns>
    <mx:DataGridColumn dataField="Album" width="200" />
    <mx:DataGridColumn dataField="Price" headerText="List Price" />
</mx:columns>
```

Hiding and displaying columns

If you might display a column at some times, but not at others, you can specify the DataGridColumn class `visible` property to hide or show the column. The following example lets you hide or show the album price by clicking a button:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridVisibleColumn.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

   <mx:DataGrid id="myDG" width="350">
      <mx:ArrayList>
          <mx:source>
              <fx:Object Artist="Pavement" Price="11.99"
                  Album="Slanted and Enchanted" />
              <fx:Object Artist="Pavement"
                  Album="Brighten the Corners" Price="11.99" />
          </mx:source>
      </mx:ArrayList>
      <mx:columns>
         <mx:DataGridColumn dataField="Artist" />
         <mx:DataGridColumn dataField="Album" />
         <mx:DataGridColumn id="price" dataField="Price" visible="false"/>
      </mx:columns>
   </mx:DataGrid>
   <!-- The column id property specifies the column to show.-->
   <mx:Button label="Toggle Price Column"
      click="price.visible = !price.visible;" />
</s:Application>
```

Passing data to an MX DataGrid control

Flex lets you populate a DataGrid control from an ActionScript variable definition or from a Flex data model. The following example populates a DataGrid control by using a variable:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridPassData.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
   initialize="initData();">

    <fx:Script>
        <![CDATA[
            import mx.collections.*;
            private var DGArray:Array = [
                {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}];

            [Bindable]
            public var initDG:ArrayList;
            //Initialize initDG ArrayList variable from the Array.
            //If you use this technique to process an HTTPService,
            //WebService, or RemoteObject result, use an ArrayCollection
            //rather than an ArrayList.
            public function initData():void {
                initDG=new ArrayList(DGArray);
            }
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" width="350" height="200"
        dataProvider="{initDG}" >
        <mx:columns>
            <mx:DataGridColumn dataField="Album" />
            <mx:DataGridColumn dataField="Price" />
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you bind the variable `initDG` to the `<mx:dataProvider>` property. You can still specify a column definition event when using data binding. For a description of using a model as a data provider, see “Populating a ComboBox control by using variables and models” on page 960

Handling events in an MX DataGrid control

The DataGrid control and the DataGridEvent class define several event types that let you respond to user interaction. For example, Flex broadcasts mx.events.ListEvent class event with a `type` property value of `mx.events.ListEvent.ITEM_CLICK` ("itemClick") when a user clicks an item in a DataGrid control. You can handle this event as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.events.ListEvent;
            private function itemClickEvent(event:ListEvent):void {
                clickColumn.text=String(event.columnIndex);
                clickRow.text=String(event.rowIndex);
                eventType.text=event.type;
            }
        ]]>
    </fx:Script>

    <mx:DataGrid id="myGrid" width="350" height="150"
        itemClick="itemClickEvent(event);">
        <mx:ArrayList>
            <fx:Object Artist="Pavement" Price="11.99"
                Album="Slanted and Enchanted" />
            <fx:Object Artist="Pavement" Album="Brighten the Corners"
                Price="11.99" />
        </mx:ArrayList>
    </mx:DataGrid>
    <s:Form>
        <s:FormItem label="Column Index:">
            <s:Label id="clickColumn"/>
        </s:FormItem>
        <s:FormItem label="Row Index:">
            <s:Label id="clickRow"/>
        </s:FormItem>
        <s:FormItem label="Type:">
            <s:Label id="eventType"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

In this example, you use the event handler to display the column index, row index, and event type in three TextArea controls.

The index of columns in the DataGrid control is zero-based, meaning values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of columns. Row items are also indexed starting at 0. Therefore, if you select the first item in the second row, this example displays 0 in the TextArea for the column index, and 1 in the TextArea for the row index.

To access the selected item in the event handler, you can use the `currentTarget` property of the event object, and the `selectedItem` property of the DataGrid control, as the following code shows:

```
var selectedArtist:String=event.currentTarget.selectedItem.Artist;
```

The `currentTarget` property of the object passed to the event handler contains a reference to the DataGrid control. You can reference any control property by using `currentTarget` followed by a period and the property name. The `currentTarget.selectedItem` field contains the selected item.

Sorting data in an MX DataGrid control

The DataGrid control supports displaying sorted data in two ways:

- By default, the control displays data in the sorted order of its underlying data provider collection. Therefore you can use the collection Sort and SortField classes to control the order of the rows.
- By default, users can sort the display by clicking the column headers. Clicking the column header initially sorts the display in descending order of the entries in the selected column, and clicking the header again reverses the sort order. You can disable sorting an entire DataGrid Control or individual columns.

  For detailed information on using the Sort and SortField classes, see "Sorting and filtering data for viewing" on page 912.

Determining the initial DataGrid sort order

To specify the initial DataGrid sort order, you sort the data provider. While a number of approaches can work, the following technique takes best advantage of the built in features of Flex collections:

- Use an object that implements the ICollectionView interface, such as an ArrayCollection, in the `dataProvider` property of your DataGrid. Specify a Sort object in the data provider object's the `sort` field. Note that you cannot use an ArrayList with a sort field.
- Use the Sort object to control the order of the rows in the `dataProvider` object.

Controlling user sorting of DataGrid displays

The following DataGrid and DataGridColumn properties control how users can sort data:

- The DataGrid `sortableColumns` property is a global switch that enables user sorting of the DataGrid display by clicking column headings. The default this property is `true`.
- The DataGridColumn`sortable` property specifies whether users can sort an individual column. The default this property is `true`.
- The DataGridColumn `sortCompareFunction` property lets you specify a custom comparison function. This property sets the `compare` property of the default SortField class object that the DataGrid uses to sort the grid when users click the headers. It lets you specify the function that compares two objects and determines which would be higher in the sort order, without requiring you to explicitly create a Sort object on your data provider. For detailed information on the comparison function signature and behavior, see sortCompareFunction in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

  By default, the DataGrid class uses its own sort code to control how the data gets sorted when the user clicks a column. To override this behavior, you create a `headerRelease` event handler to handle the DataGridEvent event that is generated when the user clicks the column header. This event handler must do the following:

  1. Use the event object's `columnIndex` property to determine the clicked column.
  2. Create an MX Sort object with a set of MX SortField objects based on the clicked column and any other rules that you need to control the sorting order. For more information on using Sort objects, see "Sorting and filtering data for viewing" on page 912.
  3. Apply the Sort object to the collection assigned as the data provider.
  4. Call the DataGridEvent class event object's `preventDefault()` method to prevent the DataGrid from doing a default column sort.

     ***Note:** If you specify a* `labelFunction` *property, you must also specify a* `sortCompareFunction` *function. The Computed Columns example in Flex Explorer shows this use.*

The following example shows how to use the headerRelease event handler to do multi-column sorting when a user clicks a DataGrid column header.

Example: Sorting a DataGrid on multiple columns

The following example shows how you can use a collection with a Sort object to determine an initial multi-column sort and to control how the columns sort when you click the headers. The data grid is initially sorted by in-stock status first, artist second, and album name, third. If you click any heading, that column becomes the primary sort criterion, the previous primary criterion becomes the second criterion, and the previous secondary criterion becomes the third criterion.

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSort.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initDP();"
    width="550" height="400">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.events.DataGridEvent;
            import mx.collections.*;

            // Declare storage variables and initialize the simple variables.
            // The data provider collection.
            private var myDPColl:ArrayCollection;
            // The Sort object used to sort the collection.
            [Bindable]
            private var sortA:Sort;

            // The sort fields used to determine the sort.
            private var sortByInStock:SortField;
            private var sortByArtist:SortField;
            private var sortByAlbum:SortField;
            private var sortByPrice:SortField;
            // The data source that populates the collection.
            private var myDP:Array = [
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, InStock: true},
                {Artist:'Pavement', Album:'Crooked Rain, Crooked Rain',
                    Price:10.99, InStock: false},
                {Artist:'Pavement', Album:'Wowee Zowee',
                    Price:12.99, InStock: true},
                {Artist:'Asphalt', Album:'Brighten the Corners',
                    Price:11.99, InStock: false},
                {Artist:'Asphalt', Album:'Terror Twilight',
                    Price:11.99, InStock: true},
                {Artist:'Asphalt', Album:'Buildings Meet the Sky',
                    Price:14.99, InStock: true},
                {Artist:'Other', Album:'Other',
                    Price:5.99, InStock: true}
            ];
            //Initialize the DataGrid control with sorted data.
            private function initDP():void {
                //Create an ArrayCollection backed by the myDP array of data.
                myDPColl = new ArrayCollection(myDP);
                //Create a Sort object to sort the ArrrayCollection.
                sortA = new Sort();
                //Initialize SortField objects for all valid sort fields:
                // A true second parameter specifies a case-insensitive sort.
                // A true third parameter specifies descending sort order.
                // A true fourth parameter specifies a numeric sort.
                sortByInStock = new SortField("InStock", true, true);
                sortByArtist = new SortField("Artist", true);
                sortByAlbum = new SortField("Album", true);
                sortByPrice = new SortField("Price", true, false, true);
                // Sort the grid using the InStock, Artist, and Album fields.
                sortA.fields=[sortByInStock, sortByArtist, sortByAlbum];
                myDPColl.sort=sortA;
                // Refresh the collection view to show the sort.
                myDPColl.refresh();
                // Initial display of sort fields
                tSort0.text = "First Sort Field: InStock";
                tSort1.text = "Second Sort Field: Artist";
                tSort2.text = "Third Sort Field: Album";
                // Set the ArrayCollection as the DataGrid data provider.
                myGrid.dataProvider=myDPColl;
                // Set the DataGrid row count to the array length,
                // plus one for the header.
                myGrid.rowCount=myDPColl.length +1;
            }

            // Re-sort the DataGrid control when the user clicks a header.
            private function headRelEvt(event:DataGridEvent):void {
                // The new third priority was the old second priority.
                sortA.fields[2] = sortA.fields[1];
                tSort2.text = "Third Sort Field: " + sortA.fields[2].name;
                // The new second priority was the old first priority.
                sortA.fields[1] = sortA.fields[0];
                tSort1.text = "Second Sort Field: " + sortA.fields[1].name;
                // The clicked column determines the new first priority.
                if (event.columnIndex==0) {
                    sortA.fields[0] = sortByArtist;
                } else if (event.columnIndex==1) {
                    sortA.fields[0] = sortByAlbum;
                } else if (event.columnIndex==2) {
                    sortA.fields[0] = sortByPrice;
                } else {
                    sortA.fields[0] = sortByInStock;}
                tSort0.text = "First Sort Field: " + sortA.fields[0].name;
                // Apply the updated sort fields and re-sort.
                myDPColl.sort=sortA;
                // Refresh the collection to show the sort in the grid.
                myDPColl.refresh();
                // Prevent the DataGrid from doing a default column sort.
                event.preventDefault();
            }
        ]]>
    </fx:Script>
    <!-- The Data Grid control.
         By default the grid and its columns can be sorted by clicking.
         The headerRelease event handler overrides the default sort
         behavior. -->
    <mx:DataGrid id="myGrid" width="100%" headerRelease="headRelEvt(event);">
        <mx:columns>
            <mx:DataGridColumn minWidth="120" dataField="Artist" />
            <mx:DataGridColumn minWidth="200" dataField="Album" />
            <mx:DataGridColumn width="75" dataField="Price" />
            <mx:DataGridColumn width="75" dataField="InStock"
               headerText="In Stock"/>
        </mx:columns>
    </mx:DataGrid>
    <mx:VBox>
        <mx:Label id="tSort0" text="First Sort Field: "/>
        <mx:Label id="tSort1" text="Second Sort Field: "/>
        <mx:Label id="tSort2" text="Third Sort Field: "/>
    </mx:VBox>
</s:Application>
```

MX DataGrid control user interaction

The DataGrid control responds to mouse and keyboard activity. The response to a mouse click or key press depends on whether a cell is editable. A cell is editable when the `editable` properties of the DataGrid control and the DataGridColumn containing the cell are both `true`. Clicking within an editable cell directs focus to that cell. Clicking a noneditable cell has no effect on the focus.

Users can modify the DataGrid control appearance in the following ways:

- If the value of the `sortableColumns` property is `true`, the default value, clicking within a column header causes the DataGrid control to be sorted based on the column's cell values.
- If the value of the `draggableColumns` property is `true`, the default value, clicking and holding the mouse button within a column header, dragging horizontally, and releasing the mouse button moves the column to new location.
- If the value of the `resizableColumns` property is `true`, the default value, clicking in the area between columns permits column resizing.

Keyboard navigation

The DataGrid control has the following keyboard navigation features:

Key	Action
Enter Return Shift+Enter	When a cell is in editing state, commits change, and moves editing to the cell on the same column, next row down or up, depending on whether Shift is pressed.
Tab	Moves focus to the next editable cell, traversing the cells in row order. If at the end of the last row, advances to the next element in the parent container that can receive focus.
Shift+Tab	Moves focus to the previous editable cell. If at the beginning of a row, advances to the end of the previous row. If at the beginning of the first row, advances to the previous element in the parent container that can receive focus.
Up Arrow Home Page Up	If editing a cell, shifts the cursor to the beginning of the cell's text. If the cell is not editable, moves selection up one item.

Key	Action
Down Arrow End Page Down	If editing a cell, shifts the cursor to the end of the cell's text. If the cell is not editable, moves selection down one item.
Control	Toggle key. If you set the DataGrid control `allowMultipleSelection` property to `true`, allows for multiple (noncontiguous) selection and deselection. Works with key presses, click selection, and drag selection.
Shift	Contiguous select key. If you set the DataGrid control `allowMultipleSelection` property to `true`, allows for contiguous selections. Works with key presses, click selection, and drag selection.

MX Tree control

The Tree control lets a user view hierarchical data arranged as an expandable tree.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For information on hierarchical data providers, see "Hierarchical data objects" on page 927.

For information on the following topics, which are often important for using advanced Tree controls, see:

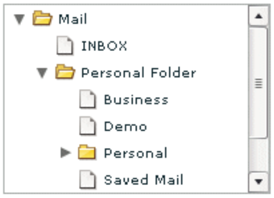
- How to format the information in each Tree node and control how users enter data in the nodes; see "MX item renderers and item editors" on page 1006.
- How to drag objects to and from the Tree control; see "Drag and drop" on page 1893.

About Tree controls

A Tree control is a hierarchical structure of *branch* and *leafnodes*. Each item in a tree is called a node and can be either a leaf or a branch. A branch node can contain leaf or branch nodes, or can be empty (have no children). A leaf node is an end point in the tree.

By default, a leaf is represented by a text label beside a file icon and a branch is represented by a text label beside a folder icon with a disclosure triangle that a user can open to expose children.

The following image shows a Tree control:



Creating a Tree control

You define a Tree control in MXML by using the `<mx:Tree>` tag. The Tree class extends the List class and Tree controls take all of the properties and methods of the List control. For more information about using the List control, see "MX List control" on page 943. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

The Tree control normally gets its data from a hierarchical data provider, such as an XML structure. If the Tree represents dynamically changing data, you should use a list or collection, such as the standard ArrayCollection, or XMLListCollection object, as the data provider.

The Tree control uses a data descriptor to parse and manipulate the data provider content. By default, the Tree control uses a DefaultDataDescriptor instance, but you can create your own class and specify it in the Tree control's `dataDescriptor` property.

The DefaultDataDescriptor class supports the following types of data:

Collections A collection implementation, such as an XMLListCollection or ArrayCollection object. The DefaultDataDescriptor class includes code to handle collections efficiently. Always use a collection as the data provider if the data in the menu changes dynamically; otherwise the Tree control might display obsolete data.

XML A string containing valid XML text, or any of the following objects containing valid E4X format XML data: `<fx:XML>` or <fx:XMLList> compile-time tag, or an XML or XMLList object.

Other objects An array of items, or an object that contains an array of items, where a node's children are contained in an item named `children`.

The DefaultDataDescriptor class also supports using an `<fx:Model>` tag as a data provider for a menu, but all leaf nodes must have the name `children`; As a general rule, it is a better programming practice to use the `<fx:XML>` or `<fx:XMLList>` tags when you need a Tree data provider that uses binding.

For more information on hierarchical objects and data descriptors, including a detailed description of the formats supported by the DefaultDataDescriptor, see "Data descriptors and hierarchical data structure" on page 927.

The following code contains a single Tree control that defines the tree shown in the image in "MX Tree control" on page 973. This uses an XMLListCollection wrapper around an `<fx:XMLList>` tag. By using an XMLListCollection, you can modify the underlying XML data provider by changing the contents of the `MailBox` XMLListCollection, and the Tree control will represent the changes to the data. This example also does not use the `<mx:dataProvider>` tag because `dataProvider` is the default property of the Tree control.

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

   <mx:Tree id="tree1"
       labelField="@label"
       showRoot="true"
       width="160">
      <mx:XMLListCollection id="MailBox">
         <fx:XMLList>
            <folder label="Mail">
               <folder label="INBOX"/>
               <folder label="Personal Folder">
                  <Pfolder label="Business" />
                  <Pfolder label="Demo" />
                     <Pfolder label="Personal" isBranch="true" />
                     <Pfolder label="Saved Mail" />
                  </folder>
                  <folder label="Sent" />
                  <folder label="Trash" />
            </folder>
         </fx:XMLList>
      </mx:XMLListCollection>
   </mx:Tree>
</s:Application>
```

The tags that represent tree nodes in the XML data can have any name. The Tree control reads the XML and builds the display hierarchy based on the nested relationship of the nodes. For information on valid XML structure, see "Hierarchical data objects" on page 927.

Some data providers have a single top level node, called a *root* node. Other data providers are lists of nodes and do not have a root node. In some cases, you might not want to display the root node as the Tree root. To prevent the tree from displaying the root node, specify the `showRoot` property to `false;` doing this does not affect the data provider contents, only the Tree display. You can only specify a `falseshowRoot` property for data providers that have roots, that is, XML and Object-based data providers.

A branch node can contain multiple child nodes, and, by default, appears as a folder icon with a disclosure triangle that lets users open and close the folder. Leaf nodes appear by default as file icons and cannot contain child nodes.

When a Tree control displays a node of a non-XML data provider, by default, it displays the value of the `label` property of the node as the text label. When you use an E4X XML-based data provider, however, you must specify the label field, even if the label is identified by an attribute named "label". To specify the label field, use the `labelField` property; for example, if the label field is the label attribute, specify `labelField="@label"`.

Handling Tree control events

You typically use events to respond to user interaction with a Tree control. Since the Tree control is derived from the List control, you can use all of the events defined for the List control. The Tree control also dispatches several Event and TreeEvent class events, including `Event.change` and `TreeEvent.itemOpen`. The following example defines event handlers for the `change` and `itemOpen` events:

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

   <fx:Script>
      <![CDATA[
         import flash.events.*;
         import mx.events.*;
         import mx.controls.*;
         private function changeEvt(event:Event):void {
            var theData:String = ""
            if (event.currentTarget.selectedItem.@data) {
               theData = " Data: " + event.currentTarget.selectedItem.@data;
            }
            forChange.text = event.currentTarget.selectedItem.@label + theData;
         }
      private function itemOpenEvt(event:TreeEvent):void {
         forOpen.text = event.item.@label;
      }
   ]]>
   </fx:Script>
   <mx:Tree id="XMLTree1" width="150" height="170"
         labelField="@label" itemOpen="itemOpenEvt(event);"
         change="changeEvt(event);">
      <mx:XMLListCollection id="MailBox">
         <fx:XMLList>
```

```
            <node label="Mail" data="100">
               <node label="Inbox" data="70"/>
               <node label="Personal Folder" data="10">
                  <node label="Business" data="2"/>
                  <node label="Demo" data="3"/>
                  <node label="Personal" data="0" isBranch="true" />
                  <node label="Saved Mail" data="5" />
               </node>
               <node label="Sent" data="15"/>
               <node label="Trash" data="5"/>
            </node>
         </fx:XMLList>
      </mx:XMLListCollection>
   </mx:Tree>
    <s:Form>
        <s:FormItem label="Change Event:">
            <s:Label id="forChange" width="150"/>
        </s:FormItem>
        <s:FormItem label="Open Event:">
            <s:Label id="forOpen" width="150"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

In this example, you define event listeners for the `change` and `itemOpen` events. The Tree control broadcasts the `change` event when the user selects a tree item, and broadcasts the `itemOpen` event when a user opens a branch node. For each event, the event handler displays the label and the data property, if any, in a TextArea control.

Expanding a tree node

By default, the Tree control displays the root node or nodes of the tree when it first opens. If you want to expand a node of the tree when the tree opens, you can use the `expandItem()` method of the Tree control. The following change to the example in "Handling Tree control events" on page 975 calls the `expandItem()` method as part of the Tree control's `creationComplete` event listener to expand the root node of the tree:

```
<fx:Script>
    <![CDATA[
.
.
.
        private function initTree():void {
            XMLTree1.expandItem(MailBox.getItemAt(0), true);
            forOpen.text=XMLTree1.openItems[0].@label;
        }
        ]]>
    </fx:Script>

<mx:Tree id="tree1" ... creationComplete="initTree();" >
    ...
</mx:Tree>
```

This example must use the Tree control's `creationComplete` event, not the `initialize` event, because the data provider is not fully initialized and available until the `creationComplete` event.

The Tree control `openItems` property is an Array containing all expanded tree nodes. The following line in the example code displays the label of the first (and only) open item in the tree:

```
forOpen.text=XMLTree1.openItems[0].@label;
```

In this example, however, you could also get the `openItems` box to indicate the initial open item by setting the `expandItem()` method to dispatch an `itemOpen` event. You can do this by specifying the fourth, optional parameter of the `expandItem()` method to `true`. The `true` fourth parameter causes the tree to dispatch an `open` event when the item opens. The following example shows the use of the fourth parameter:

```
XMLTree1.expandItem(MailBox.getItemAt(0), true, false, true);
```

You can programmatically walk down a Tree control's nodes and open a node without knowing what depth you are at in the Tree's data provider. One way to do this is by using the `children()` method of a node's XML object to test whether the node has any children; you can use the `expandItem()` method to open the node.

The following example opens nodes in the Tree control based on the values of query string parameters. For example, if you pass *app_url*?0=1&1=2&2=0 to the application, then Flex opens the second item at the top level of the tree, the third item at the next level, and the first item at the third level of the tree (nodes are zero-based).

```
<?xml version="1.0"?>
<!-- dpcontrols/DrillIntoTree.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">

    <fx:Script>
      <![CDATA[
        import mx.collections.XMLListCollection;
        import mx.core.FlexGlobals;

        [Bindable]
        private var treeData:XML =
            <root>
                <node label="Monkeys">
                    <node label="South America">
                        <node label="Coastal"/>
                        <node label="Inland"/>
                    </node>
                    <node label="Africa" isBranch="true"/>
                    <node label="Asia" isBranch="true"/>
                </node>
                <node label="Sharks">
                    <node label="South America" isBranch="true"/>
                    <node label="Africa" isBranch="true"/>
                    <node label="Asia" >
                        <node label="Coastal"/>
                        <node label="Inland"/>
                    </node>
                </node>
            </root>;

        private var openSequence:Array = [];
        private function initApp():void {
            /*  Parse URL and place values into openSequence Array.
                This lets you pass any integers on the query string,
                in any order. So:
                http://localhost/flex/flex2/DrillIntoTree.swf?0=1&1=2&2=0
                results in an array of selections like this:
                0:1
                1:2
                2:0
                Non-ints are ignored.
                The Array is then used to drill down into the tree.
            */
           var paramLength:int = 0;
           for (var s:String in FlexGlobals.topLevelApplication.parameters) {
               if (!isNaN(Number(s))) {
                    openSequence[s] = FlexGlobals.topLevelApplication.parameters[s];
                    paramLength += 1;
               }
            }
            openTree();
        }

        private function openTree():void {
            var nodeList:XMLListCollection =
                myTree.dataProvider as XMLListCollection;
            var node:XMLList = nodeList.source;
            for(var i:int=0; i < openSequence.length; i++) {
                var j:int = openSequence[i];
                var n:XML = node[j];
                if( n.children() != null ) {
                    myTree.expandItem(n,true,false);
                    node = n.children();
                } else {
                    break;
                }
            }
            if( n != null ) myTree.selectedItem = n;
        }
      ]]>
    </fx:Script>

    <mx:Tree id="myTree"
        y="50"
        width="221"
        height="257"
        horizontalCenter="0"
        dataProvider="{treeData.node}"
        labelField="@label"/>
</s:Application>
```

Specifying Tree control icons

The Tree control provides four techniques for specifying node icons:

- The `folderOpenIcon`, `folderClosedIcon`, and `defaultLeafIcon` properties
- Data provider node icon fields
- The `setItemItcon()` method
- The `iconFunction` property

Using icon properties

You can use the `folderOpenIcon`, `folderClosedIcon`, and `defaultLeafIcon` properties to control the Tree control icons. For example, the following code specifies a default leaf icon, and icons for the open and closed states of branch nodes:

```
<mx:Tree folderOpenIcon="@Embed(source='open.jpg')"
    folderClosedIcon="@Embed(source='closed.jpg')"
    defaultLeafIcon="@Embed(source='def.jpg')">
```

Using icon fields

You can specify an icon displayed with each Tree leaf when you populate it by using XML, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeIconField.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            [Bindable]
            [Embed(source="assets/radioIcon.jpg")]
            public var iconSymbol1:Class;
            [Bindable]
            [Embed(source="assets/topIcon.jpg")]
            public var iconSymbol2:Class;
        ]]>
    </fx:Script>
   <mx:Tree iconField="@icon"
       labelField="@label"
       showRoot="false"
        width="160">
        <fx:XMLList>
            <node label="New">
                <node label="HTML Document" icon="iconSymbol2"/>
                <node label="Text Document" icon="iconSymbol2"/>
            </node>
            <node label="Close" icon="iconSymbol1"/>
        </fx:XMLList>
    </mx:Tree>
</s:Application>
```

In this example, you use the `iconField` property to specify the field of each item containing the icon. You use the `Embed` metadata to import the icons, then reference them in the XML definition. You cannot use icon fields to specify icons for individual branch nodes; instead you must use the Tree control's `folderOpenIcon`, `folderClosedIcon` properties, each of which specifies an icon to use for all open or closed branches, or use the `setItemIcon()` method to set individual node icons.

Using the setItemIcon() method

You can use the `setItemIcon()` method to specify the icon, or both the open and closed icons for a tree item. This method lets you dynamically specify and change icons for individual branches and nodes. For details on this function see setItemIcon() in *ActionScript 3.0 Reference for the Adobe Flash Platform*. The following example sets the open and closed node icon for the first branch node and the icon for the second branch (which does not have any leaves):

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeItemIcon.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
   <fx:Script>
      <![CDATA[
         [Bindable]
         [Embed(source="assets/radioIcon.jpg")]
         public var iconSymbol1:Class;
         [Bindable]
         [Embed(source="assets/topIcon.jpg")]
         public var iconSymbol2:Class;

         private function setIcons():void {
            myTree.setItemIcon(myTree.dataProvider.getItemAt(0),
               iconSymbol1, iconSymbol2);
            myTree.setItemIcon(myTree.dataProvider.getItemAt(1),
               iconSymbol2, null);
         }
      ]]>
   </fx:Script>
   <mx:Tree id="myTree" labelField="@label"
      showRoot="false"
      width="160" initialize="setIcons();">
      <fx:XMLList>
         <node label="New">
            <node label="HTML Document"/>
            <node label="Text Document"/>
         </node>
         <node label="Close"/>
      </fx:XMLList>
   </mx:Tree>
</s:Application>
```

Using an icon function

You can use the Tree control `iconFunction` property to specify a function that dynamically sets all icons for the tree. For information on using the `iconFunction` property in Flex controls, see “Specifying an icon to the List control” on page 948.

Opening a Tree control to a specific node

By default, a Tree control is collapsed when it initializes, but you can initialize it so that it is expanded with a specific node selected, as the following example shows. In this application, the `initTree()` method is called after the Tree control is created. This method expands the root node of the Tree control and sets its `selectedIndex` property to the index number of a specific node.

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeOpenNode.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import flash.events.*;
            import mx.events.*;
            import mx.controls.*;
            private function initTree():void {
                XMLTree1.expandItem(MailBox.getItemAt(0), true);
                XMLTree1.selectedIndex = 2;
            }
        ]]>
    </fx:Script>
    <mx:Tree id="XMLTree1"
        width="150" height="170"
        labelField="@label"
        creationComplete="initTree();">
        <mx:XMLListCollection id="MailBox">
            <fx:XMLList>
                <node label="Mail" data="100">
                    <node label="Inbox" data="70"/>
                    <node label="Personal Folder" data="10">
                        <node label="Business" data="2"/>
                        <node label="Demo" data="3"/>
                        <node label="Saved Mail" data="5" />
                    </node>
                    <node label="Sent" data="15"/>
                    <node label="Trash" data="5"/>
                   </node>
            </fx:XMLList>
        </mx:XMLListCollection>
    </mx:Tree>
</s:Application>
```

Adding and removing leaf nodes at run time

You can add and remove leaf nodes from a Tree control at run time. The following example contains code for making these changes. The application initializes with predefined branches and leaf nodes that represent company departments and employees. You can dynamically add leaf (employee) nodes to the Operations department branch at run time. You can also remove any leaf node, including any you create at run time.

The XML in this example contains two different element names, department and employee. The Tree control's label function, `treeLabel()`, determines what text is displayed for these types of elements. It uses E4X syntax to return either the title of a department or the name of an employee. Those values are then used in the `addEmployee()`and `removeEmployee()`methods.

To add employees to the Operations department, the `addEmployee()` method uses E4X syntax to get the Operations department node based on the value of its title attribute, and stores it in the dept variable, which is of type XMLList. It then appends a child node to the Operations node by calling `dept.appendChild()`.

The `removeEmployee()` method stores the currently selected item in the node variable, which is of type XML. The method calls the `node.localName()` method to determine if the selected item is an employee node. If the item is an employee node, it is deleted.

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeAddRemoveNode.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.collections.XMLListCollection;

            [Bindable]
            private var company:XML =
              <list>
                <department title="Finance" code="200">
                    <employee name="John H"/>
                    <employee name="Sam K"/>
                </department>
                <department title="Operations" code="400">
                    <employee name="Bill C"/>
                    <employee name="Jill W"/>
                </department>
                <department title="Engineering" code="300">
                    <employee name="Erin M"/>
                    <employee name="Ann B"/>
                </department>
              </list>;

            [Bindable]
            private var companyData:XMLListCollection =
                new XMLListCollection(company.department);

            private function treeLabel(item:Object):String {
                var node:XML = XML(item);
                if( node.localName() == "department" )
                    return node.@title;
                else
                    return node.@name;
            }
            private function addEmployee():void {
                var newNode:XML = <employee/>;
                newNode.@name = empName.text;
                var dept:XMLList =company.department.(@title == "Operations");
                if( dept.length() > 0 ) {
                    dept[0].appendChild(newNode);
                    empName.text = "";
                }
            }
            private function removeEmployee():void {
                var node:XML = XML(tree.selectedItem);
                if( node == null ) return;
                if( node.localName() != "employee" ) return;
                var children:XMLList = XMLList(node.parent()).children();
                for(var i:Number=0; i < children.length(); i++) {
                    if( children[i].@name == node.@name ) {
                        delete children[i];
                    }
                }
            }
        ]]>
    </fx:Script>

    <mx:Tree id="tree"
        top="72" left="50"
        dataProvider="{companyData}"
        labelFunction="treeLabel"
        height="225" width="300"/>

    <mx:VBox>
        <mx:HBox>
            <mx:Button label="Add Operations Employee" click="addEmployee();"/>
            <mx:TextInput id="empName"/>
        </mx:HBox>
      <mx:Button label="Remove Selected Employee" click="removeEmployee();"/>
    </mx:VBox>
</s:Application>
```

Tree user interaction

You can let users edit tree control labels. The controls also support several keyboard navigation and editing keys.

Editing a node label at run time

Set the `editable` property of the Tree control to `true` to make node labels editable at run time. To edit a node label, the user selects the label, and then enters a new label or edits the existing label text.

To support label editing, the Tree control's List superclass uses the following events. These events belong to the ListEvent class:

Event	Description
`itemEditBegin`	Dispatched when the editedItemPosition property has been set and the cell can be edited.
`itemEditEnd`	Dispatched when cell editing session ends for any reason.
`itemFocusIn`	Dispatched when tree node gets the focus: when a user selects the label or tabs to it.
`itemFocusOut`	Dispatched when a label loses focus.
`itemClick`	Dispatched when a user clicks on an item in the control.

These events are commonly used in custom item editors. For more information see "MX item renderers and item editors" on page 1006.

Using the keyboard to edit labels

If you set the Tree `editable` property to `true`, you can use the following keys to edit labels:

Key	**Description**
Down Arrow Page Down End	Moves the caret to the end of the label.
Up Arrow Page Up Home	Moves the caret to the beginning of the label.
Right Arrow	Moves the caret forward one character.
Left Arrow	Moves the caret backward one character.
Enter	Ends editing and moves selection to the next visible node, which can then be edited. At the last node, selects the label.
Shift Enter	Ends editing and moves selection to the previous visible node, which can then be edited. At the first node, selects the label.
Escape	Cancels the edit, restores the text, and changes the row state from editing to selected.
TAB	When in editing mode, accepts the current changes, selects the row below, and goes into editing mode with the label text selected. If at the last element in the tree or not in editing mode, sends focus to the next control.
Shift-TAB	When in editing mode, accepts the current changes, selects the row above, and goes into editing mode. If at the first element in the tree or not in editing mode, sends focus to the previous control.

Tree Navigation keys

When a Tree control is not editable and has focus from clicking or tabbing, you use the following keys to control it:

Key	**Description**
Down Arrow	Moves the selection down one. When the Tree control gets focus, use the Down arrow to move focus to the first node.
Up Arrow	Moves the selection up one item.
Right Arrow	Opens a selected branch node. If a branch is already open, moves to the first child node.
Left Arrow	Closes a selected branch node. If a leaf node or a closed branch node is currently selected, selects the parent node.
Spacebar or * (Asterisk on numeric keypad)	Opens or closes a selected branch node (toggles the state).
+ (Plus sign on numeric keypad)	Opens a selected branch node.
- (Minus sign on numeric keypad)	Closes a selected branch node.
Control + Arrow keys	Moves the focus, but does not select a node. Use the Spacebar to select a node.
End	Moves the selection to the bottom of the list.
Home	Moves the selection to the top of the list.
Page down	Moves the selection down one page.

Key	Description
Page up	Moves the selection up one page.
Control	If the `allowMultipleSelection` property is `true`, allows multiple noncontiguous selections.
Shift	If the `allowMultipleSelection` property is `true`, allows multiple contiguous selections.

Menu-based controls

There are three Adobe® Flex® Framework controls that you can use to create or interact with menus.

About menu-based controls

Flex framework includes three controls that present hierarchical data in a cascading menu format. All menu controls can have icons and labels for each menu item, and dispatch events of the mx.events.MenuEvent class in response to user actions. You can use the following menu-based controls:

Menu control A visual menu that can have cascading submenus. You typically display a menu control in response to some user action, such as clicking a button. You cannot define a Menu control by using an MXML tag; you must define it, display it, and hide it by using ActionScript.

MenuBar control A horizontal bar of menu items. Each item in the menu bar can have a submenu that displays when you click the MenuBar item. The MenuBar control is effectively a static (non-pop-up) menu that displays the top level of the menu as the bar items.

PopUpMenuButton control A subclass of the PopUpButton control that displays a Menu control when you click the secondary button. The primary button label changes when you select an item from the pop-up menu.

Defining menu structure and data

All menu-based controls use data providers with the following characteristics to specify the structure and contents of the menus:

- The data providers are often hierarchical, but you can also have a single-level menu.
- Individual menu items include fields that determine the menu item appearance and behavior. Menu-based controls support fields that define the label text, an icon, the menu item type, and item status. For information on meaningful fields, see "Specifying and using menu entry information" on page 986.

About menu data providers

The `dataProvider` property of a menu-based control specifies an object that defines the structure and contents of the menu. If a menu's contents are dynamic, you change the menu by modifying its data provider.

Menu-based controls typically get their data from hierarchical data providers, such as nested arrays of objects or XML. If the menu represents dynamically changing data, you use an object that implements the ICollectionView interface, such as ArrayCollection or XMLListCollection.

Menu-based controls use a data descriptor to parse and manipulate the data provider contents. By default, menu-based controls use a DefaultDataDescriptor class descriptor, but you can create your own class and specify it in the Menu control's `dataDescriptor` property.

The DefaultDataDescriptor class supports the following types of data:

XML A string that contains valid XML text, or any of the following objects containing valid E4X format XML data: an `<fx:XML>` or `<fx:XMLList>` compile-time tag, or an XML or XMLList object.

Other objects An array of items, or an object that contains an array of items, where a node's children are contained in an item named `children`. You can also use the `<fx:Model>` compile-time tag to create nested objects that support data binding, but you must follow the structure defined in "Using the <fx:Model> tag with Tree and menu-based controls" on page 929.

Collections An object that implements the ICollectionView interface (such as the ArrayCollection or XMLListCollection classes) and whose data source conforms to the structure specified in either of the previous bullets. The DefaultDataDescriptor class includes code to handle collections efficiently. Always use a collection as the data provider if the data in the menu changes dynamically; otherwise, the Menu displays obsolete data.

For more information on hierarchical objects and data descriptors, including a detailed description of the formats supported by the DefaultDataDescriptor, see "Data descriptors and hierarchical data structure" on page 927.

As with all data-driven controls, if the data provider contents can change dynamically, and you want the Menu to update with the changes, ensure that the data source is a collection, such as an ArrayCollection or XMLListCollection object. To modify the menu, change the underlying collection, and the menu will update its appearance accordingly.

Node (menu item) tags in the XML data can have any name. Many examples in this topic use tags such as `<node>` for all menu items, or `<menuItem>` for top-level items and `<subMenuItem>` for submenu items, but it might be more realistic to use tag names that identify the data, such as `<person>`, `<address>`, and so on. The menu-handling code reads through the XML and builds the display hierarchy based on the nested relationship of the nodes. For more information, see "Specifying and using menu entry information" on page 986.

Most menus have multiple items at the top level, not a single root item. XML objects, such as the XML object created by the `<fx:XML>` tag, must have a single root node. To display a menu that uses a data provider that has a root that you do not want to display, set the Menu, PopUpMenuButton, or MenuBar `showRoot` property to `false`.

Specifying and using menu entry information

Information in menu-based control data providers determines how each menu entry appears and is used. To access or change the menu contents, you modify the contents of the data provider.

The menu-based classes use the methods of the IMenuDataDescriptor interface to access and manipulate information in the data provider that defines the menu behavior and contents. Flex provides a DefaultDataDescriptor class that implements this interface. The menu-based controls use the DefaultDataDescriptor class if you do not set the `dataDescriptor` property.

Menu entry types

Each data provider entry can specify an item type and type-specific information about the menu item. Menu-based classes support the following item types (`type` field values):

normal (the default) Selecting an item with the `normal` type triggers a `change` event, or, if the item has children, opens a submenu.

check Selecting an item with the `check` type toggles the menu item's `toggled` property between `true` and `false` values. When the menu item is in the `true` state, it displays a check mark in the menu next to the item's label.

radio Items with the `radio` type operate in groups, much like RadioButton controls; you can select only one radio menu item in each group at a time. The example in this section defines three submenu items as radio buttons within the group "one".

When a radio button is selected, the radio item's toggled property is set to true, and the toggled properties of all other radio items in the group are set to false. The Menu control displays a solid circle next to the radio button that is currently selected. The selection property of the radio group is set to the label of the selected menu item.

separator Items with the separator type provide a simple horizontal line that divides the items in a menu into different visual groups.

Menu attributes

Menu items can specify several attributes that determine how the item is displayed and behaves. The following table lists the attributes you can specify, their data types, their purposes, and how the data provider must represent them if the menu uses the DefaultDataDescriptor class to parse the data provider:

Attribute	Type	Description
enabled	Boolean	Specifies whether the user can select the menu item (true), or not (false). If not specified, Flex treats the item as if the value were true. If you use the default data descriptor, data providers must use an enabled XML attribute or object field to specify this characteristic.
groupName	String	(Required, and meaningful, for radio type only) The identifier that associates radio button items in a radio group. If you use the default data descriptor, data providers must use a groupName XML attribute or object field to specify this characteristic.
icon	Class	Specifies the class identifier of an image asset. This item is not used for the check, radio, or separator types. You can use the checkIcon and radioIcon styles to specify the icons used for radio and check box items that are selected. The menu's iconField or iconFunction property determines the name of the field in the data that specifies the icon, or a function for determining the icons.
label	String	Specifies the text that appears in the control. This item is used for all menu item types except separator. The menu's labelField or labelFunction property determines the name of the field in the data that specifies the label, or a function for determining the labels. (If the data provider is in E4X XML format, you must specify one of these properties to display a label.) If the data provider is an array of strings, Flex uses the string value as the label.
toggled	Boolean	Specifies whether a check or radio item is selected. If not specified, Flex treats the item as if the value were false and the item is not selected. If you use the default data descriptor, data providers must use a toggled XML attribute or object field to specify this characteristic.
type	String	Specifies the type of menu item. Meaningful values are separator, check, or radio. Flex treats all other values, or nodes with no type entry, as normal menu entries. If you use the default data descriptor, data providers must use a type XML attribute or object field to specify this characteristic.

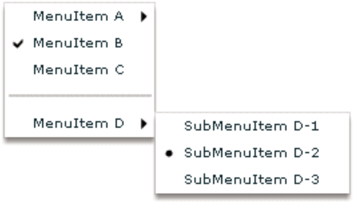
Menu-based controls ignore all other object fields or XML attributes, so you can use them for application-specific data.

Example: An Array menu data provider

The following example displays a Menu that uses an Array data provider and shows how you define the menu characteristics in the data provider. For an application that specifies an identical menu structure in XML, see "Example: Creating a simple Menu control" on page 999.

```
<?xml version="1.0"?>
<!-- menus/ArrayDataProvider.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.controls.Menu;
            // Method to create an Array-based menu.
            private function createAndShow():void {
                // The third parameter sets the showRoot property to false.
                // You must set this property in the createMenu method,
                // not later.
                var myMenu:Menu = Menu.createMenu(null, menuData, true);
                myMenu.show(10, 10);
            }
            // The Array data provider
            [Bindable]
            public var menuData:Array = [
                {label: "MenuItem A", children: [
                    {label: "SubMenuItem A-1", enabled: false},
                    {label: "SubMenuItem A-2", type: "normal"}
                    ]},
                {label: "MenuItem B", type: "check", toggled: true},
                {label: "MenuItem C", type: "check", toggled: false},
                {type: "separator"},
                {label: "MenuItem D", children: [
                    {label: "SubMenuItem D-1", type: "radio",
                        groupName: "g1"},
                    {label: "SubMenuItem D-2", type: "radio",
                        groupName: "g1", toggled: true},
                    {label: "SubMenuItem D-3", type: "radio",
                        groupName: "g1"}
                    ]}
                ];
        ]]>
    </fx:Script>
    <!-- Button control to create and open the menu. -->
    <mx:Button x="300" y="10"
        label="Open Menu"
        click="createAndShow();"/>
</s:Application>
```

The following image shows the resulting control, with MenuItem D open; notice that check item B and radio item D-2 are selected:



Example: An XML menu data provider with icons

The following example displays a menu control that uses XML as the data provider, and specifies custom icons for the items in the control:

```
<?xml version="1.0"?>
<!-- SimpleMenuControlIcon.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            // Import the Menu control.
            import mx.controls.Menu;

            [Bindable]
            [Embed(source="assets/topIcon.jpg")]
            public var myTopIcon:Class;

            [Bindable]
            [Embed(source="assets/radioIcon.jpg")]
            public var myRadioIcon:Class;

            [Bindable]
            [Embed(source="assets/checkIcon.gif")]
            public var myCheckIcon:Class;
            // Create and display the Menu control.
            private function createAndShow():void {
                var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
                myMenu.labelField="@label";
                // Specify the check icon.
                myMenu.setStyle('checkIcon', myCheckIcon);
                // Specify the radio button icon.
                myMenu.setStyle('radioIcon', myRadioIcon);
                // Specify the icon for the topmenu items.
                myMenu.iconField="@icon";
                myMenu.show(10, 10);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define the menu data. -->
        <fx:XML format="e4x" id="myMenuData">
            <root>
                <menuitem label="MenuItem A" icon="myTopIcon">
                    <menuitem label="SubMenuItem A-1" enabled="false"/>
                    <menuitem label="SubMenuItem A-2"/>
                </menuitem>
                <menuitem label="MenuItem B" type="check" toggled="true"/>
                <menuitem label="MenuItem C" type="check" toggled="false"
```

```
                    icon="myTopIcon"/>
                <menuitem type="separator"/>
                <menuitem label="MenuItem D" icon="myTopIcon">
                    <menuitem label="SubMenuItem D-1" type="radio"
                        groupName="one"/>
                    <menuitem label="SubMenuItem D-2" type="radio"
                        groupName="one" toggled="true"/>
                    <menuitem label="SubMenuItem D-3" type="radio"
                        groupName="one"/>
                </menuitem>
            </root>
        </fx:XML>
    </fx:Declarations>
    <mx:VBox>
        <!-- Define a Button control to open the menu -->
        <mx:Button id="myButton"
            label="Open Menu"
            click="createAndShow();"/>
    </mx:VBox>
</s:Application>
```

Menu-based control events

User interaction with a Menu or menu-based control is event-driven; that is, applications typically handle events generated when the user opens, closes, or selects within a menu, or submenu or rolls over or out of menu items. For detailed information on events and how to use them, see “Events” on page 54.

The Menu and MenuBar controls dispatch an identical set of menu-specific events. Event handling with PopUpMenuButton controls differs from the other two controls, but shares many elements in common with the others.

Menu control events

The Menu control defines the following menu-specific event types, of the MenuEvent class:

change (`MenuEvent.CHANGE`) Dispatched when a user changes current menu selection by using the keyboard or mouse.

itemClick (`MenuEvent.ITEM_CLICK`) Dispatched when a user selects an enabled menu item of type `normal`, `check`, or `radio`. This event is not dispatched when a user selects a menu item of type `separator`, a menu item that opens a submenu, or a disabled menu item.

itemRollOut (`MenuEvent.ITEM_ROLL_OUT`) Dispatched when the mouse pointer rolls off of a Menu item.

itemRollOver (`MenuEvent.ITEM_ROLL_OVER`) Dispatched when the mouse pointer rolls onto a Menu item.

menuHide (`MenuEvent.MENU_HIDE`) Dispatched when the entire menu or a submenu closes.

menuShow (`MenuEvent.MENU_SHOW`) Dispatched when the entire menu or a submenu opens.

The event object passed to the event listener is of type MenuEvent and contains one or more of the following menu-specific properties:

Property	Description
`item`	The item in the data provider for the menu item associated with the event.
`index`	The index of the item in the menu or submenu that contains it.
`label`	The label of the item.
`menu`	A reference to the Menu control where the event occurred.
`menuBar`	The MenuBar control instance that is the parent of the menu, or `undefined` when the menu does not belong to a MenuBar. For more information, see "MenuBar control" on page 1001.

To access properties and fields of an object-based menu item, you specify the menu item field name, as follows:

```
ta1.text = event.item.label
```

To access attributes of an E4X XML-based menu item, you specify the menu item attribute name in E4X syntax, as follows:

```
ta1.text = event.item.@label
```

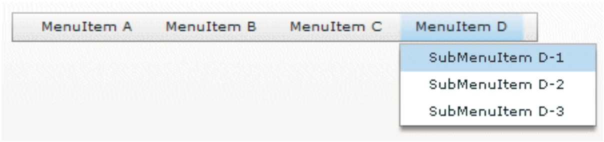
***Note:** If you set an event listener on a submenu of a menu-based control, and the menu data provider's structure changes (for example, an element is removed), the event listener might no longer exist. To ensure that the event listener is available when the data provider structure changes, either listen on events of the menu-based control, not a submenu, or add the event listener each time an event occurs that changes the data provider's structure.*

The following example shows a menu with a simple event listener. For a more complex example, see "Example: Using Menu control events" on page 994.

```
<?xml version="1.0"?>
<!-- menus/EventListener.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.controls.Menu;
            import mx.events.MenuEvent;

            // Function to create and show a menu.
            private function createAndShow():void {
                var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
                myMenu.labelField="@label"
                // Add an event listener for the itemClick event.
                myMenu.addEventListener(MenuEvent.ITEM_CLICK,
                    itemClickInfo);
                // Show the menu.
                myMenu.show(25, 10);
            }

            // The event listener for the itemClick event.
            private function itemClickInfo(event:MenuEvent):void {
                ta1.text="event.type: " + event.type;
                ta1.text+="\nevent.index: " + event.index;
                ta1.text+="\nItem label: " + event.item.@label
                ta1.text+="\nItem selected: " + event.item.@toggled;
                ta1.text+= "\nItem type: " + event.item.@type;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- The XML-based menu data provider. -->
        <fx:XML id="myMenuData">
            <xmlRoot>
                <menuitem label="MenuItem A" >
                    <menuitem label="SubMenuItem A-1" enabled="false"/>
                    <menuitem label="SubMenuItem A-2"/>
                </menuitem>
                <menuitem label="MenuItem B" type="check" toggled="true"/>
                <menuitem label="MenuItem C" type="check" toggled="false"/>
                <menuitem type="separator"/>
                <menuitem label="MenuItem D" >
                    <menuitem label="SubMenuItem D-1" type="radio"
                        groupName="one"/>
                    <menuitem label="SubMenuItem D-2" type="radio"
                        groupName="one" toggled="true"/>
                    <menuitem label="SubMenuItem D-3" type="radio"
                        groupName="one"/>
                </menuitem>
            </xmlRoot>
        </fx:XML>
    </fx:Declarations>
    <!-- Button controls to open the menus. -->
    <mx:Button
        label="Open Menu"
        click="createAndShow();"/>
    <!-- Text area to display the event information -->
    <mx:TextArea
        width="200" height="100"
        id="ta1"/>
</s:Application>
```

MenuBar events

The following figure shows a MenuBar control:



For the menu bar, the following events occur:

change (`MenuEvent.CHANGE`) Dispatched when a user changes current menu bar selection by using the keyboard or mouse. This event is also dispatched when the user changes the current menu selection in a pop-up submenu. When the event occurs on the menu bar, the `menu` property of the MenuEvent object is `null`.

itemRollOut (`MenuEvent.ITEM_ROLL_OUT`) Dispatched when the mouse pointer rolls off of a menu bar item.

itemRollOver (`MenuEvent.ITEM_ROLL_OVER`) Dispatched when the mouse pointer rolls onto a menu bar item.

menuHide (`MenuEvent.MENU_HIDE`) Dispatched when a pop-up submenu closes.

menuShow (`MenuEvent.MENU_SHOW`) Dispatched when a pop-up submenu opens, or the user selects a menu bar item with no drop-down menu.

***Note:** The MenuBar control does not dispatch the `itemClick` event when you select an item on the menu bar; it only dispatches the `itemClick` event when you select an item on a pop-up submenu.*

For each pop-up submenu, the MenuBar dispatches the `change`, `itemClick`, `itemRollOut`, `itemRollOver`, `menuShow`, and `menuHide` events in the same way it does for the Menu control. Handle events triggered by the pop-up menus as you would handle events from Menu controls. For more information, see “Menu control events” on page 990.

The following example handles events for the menu bar and for the pop-up submenus.

```
<?xml version="1.0"?>
<!-- menus/MenuBarEventInfo.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initCollections();">

    <fx:Script>
        <![CDATA[
            import mx.events.MenuEvent;
            import mx.controls.Alert;
            import mx.collections.*;
            [Bindable]
            public var menuBarCollection:XMLListCollection;

            private var menubarXML:XMLList =<>
                <menuitem label="Menu1">
                    <menuitem label="MenuItem 1-A" data="1A"/>
                    <menuitem label="MenuItem 1-B" data="1B"/>
                </menuitem>
                <menuitem label="Menu2">
                    <menuitem label="MenuItem 2-A" data="2A"/>
                    <menuitem label="MenuItem 2-B" data="2B"/>
                </menuitem>
                <menuitem label="Menu3" data="M3"/>
                </>

            // Event handler to initialize the MenuBar control.
            private function initCollections():void {
                menuBarCollection = new XMLListCollection(menubarXML);
            }

            // Event handler for the MenuBar control's change event.
            private function changeHandler(event:MenuEvent):void  {
                // Only open the Alert for a selection in a pop-up submenu.
                // The MenuEvent.menu property is null for a change event
                // dispatched by the menu bar.
                if (event.menu != null) {
                    Alert.show("Label: " + event.item.@label + "\n" +
                        "Data: " + event.item.@data, "Clicked menu item");
                }
            }
            // Event handler for the MenuBar control's itemRollOver event.
            private function rollOverHandler(event:MenuEvent):void {
                rollOverTextArea.text = "type: " + event.type + "\n";
                rollOverTextArea.text += "target menuBarIndex: " +
                    event.index + "\n";
            }
            // Event handler for the MenuBar control's itemClick event.
            private function itemClickHandler(event:MenuEvent):void {
                itemClickTextArea.text = "type: " + event.type + "\n";
                itemClickTextArea.text += "target menuBarIndex: " +
                    event.index + "\n";
            }
        ]]>
    </fx:Script>
    <mx:Panel title="MenuBar Control Example"
        height="75%" width="75%"
        paddingTop="10" paddingLeft="10">
        <mx:Label
            width="100%"
            color="blue"
            text="Select a menu item."/>
        <mx:MenuBar labelField="@label"
            dataProvider="{menuBarCollection}"
            change="changeHandler(event);"
            itemClick="itemClickHandler(event);"
            itemRollOver="rollOverHandler(event);"/>
        <mx:TextArea id="rollOverTextArea"
            width="200" height="100"/>
        <mx:TextArea id="itemClickTextArea"
            width="200" height="100"/>
    </mx:Panel>
</s:Application>
```

Example: Using Menu control events

The following example lets you experiment with Menu control events. It lets you display two menus, one with an XML data provider and one with an Array data provider. A TextArea control displays information about each event as a user opens the menus, moves the mouse, and selects menu items. It shows some of the differences in how you handle XML and object-based menus, and indicates some of the types of information that are available about each Menu event.

```
<?xml version="1.0"?>
<!-- menus/ExtendedMenuExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
        /* Import the Menu control and MenuEvent class. */
        import mx.controls.Menu;
        import mx.events.MenuEvent;

        /* Define a variable for the Menu control. */
        private var myMenu:Menu;
        /* The event listener that creates menu with an XML data
           provider and adds event listeners for the menu. */
        private function createAndShow():void {
            /* Clear the event output display. */
            ta1.text="";
            /* Don't show the (single) XML root node in the menu. */
            myMenu = Menu.createMenu(null, myMenuData, false);
            /* Set the labelField explicitly for XML data providers. */
            myMenu.labelField="@label"
            myMenu.addEventListener(MenuEvent.ITEM_CLICK, menuShowInfo);
            myMenu.addEventListener(MenuEvent.MENU_SHOW, menuShowInfo);
            myMenu.addEventListener(MenuEvent.MENU_HIDE, menuShowInfo);
            myMenu.addEventListener(MenuEvent.ITEM_ROLL_OUT, menuShowInfo);
            myMenu.addEventListener(MenuEvent.ITEM_ROLL_OVER, menuShowInfo);
            myMenu.show(275, 10);
        }
        /* The event listener for the menu events.
           Retain information on all events for a menu instance. */
        private function menuShowInfo(event:MenuEvent):void {
            ta1.text="event.type: " + event.type;
            ta1.text+="\nevent.label: " + event.label;
            /* The index value is -1 for menuShow and menuHide events. */
            ta1.text+="\nevent.index: " + event.index;
            /* The item field is null for show and hide events. */
            if (event.item) {
                ta1.text+="\nItem label: " + event.item.@label
                ta1.text+="\nItem selected: " + event.item.@toggled;
                ta1.text+= "\nItem type: " + event.item.@type;
            }
        }
        /* The event listener that creates an object-based menu
           and adds event listeners for the menu. */
        private function createAndShow2():void {
            /* Show the top (root) level objects in the menu. */
            myMenu = Menu.createMenu(null, menuData, true);
            myMenu.addEventListener(MenuEvent.ITEM_CLICK, menuShowInfo2);
            myMenu.addEventListener(MenuEvent.MENU_SHOW, menuShowInfo2);
            /* The following line is commented out so that you can see the
               results of an ITEM_CLICK event.
               (The menu hides immediately after the click.)
               myMenu.addEventListener(MenuEvent.MENU_HIDE, menuShowInfo2); */
            myMenu.addEventListener(MenuEvent.ITEM_ROLL_OVER, menuShowInfo2);
            myMenu.addEventListener(MenuEvent.ITEM_ROLL_OUT, menuShowInfo2);
            myMenu.show(275, 10);
        }
        /* The event listener for the object-based Menu events. */
        private function menuShowInfo2(event:MenuEvent):void {
            ta1.text="event.type: " + event.type;
            ta1.text+="\nevent.label: " + event.label;
            /* The index value is -1 for menuShow and menuHide events. */
            ta1.text+="\nevent.index: " + event.index;
            /* The item field is null for show and hide events. */
            if (event.item) {
                ta1.text+="\nItem label: " + event.item.label
                ta1.text+="\nItem selected: " + event.item.toggled;
                ta1.text+= "\ntype: " + event.item.type;
            }
        }
        /* The object-based data provider, an Array of objects.
           Its contents is identical to that of the XML data provider. */
        [Bindable]
        public var menuData:Array = [
            {label: "MenuItem A", children: [
                {label: "SubMenuItem A-1", enabled: false},
                {label: "SubMenuItem A-2", type: "normal"}
            ]},
            {label: "MenuItem B", type: "check", toggled: true},
            {label: "MenuItem C", type: "check", toggled: false},
            {type: "separator"},
            {label: "MenuItem D", children: [
                {label: "SubMenuItem D-1", type: "radio", groupName: "g1"},
                {label: "SubMenuItem D-2", type: "radio", groupName: "g1", toggled: true},
                {label: "SubMenuItem D-3", type: "radio", groupName: "g1"}
            ]}
        ];
    ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- The XML-based menu data provider. The XML tag requires a single root. -->
        <fx:XML id="myMenuData">
            <xmlRoot>
                <menuitem label="MenuItem A" >
                    <menuitem label="SubMenuItem A-1" enabled="false"/>
                    <menuitem label="SubMenuItem A-2"/>
                </menuitem>
                <menuitem label="MenuItem B" type="check" toggled="true"/>
                <menuitem label="MenuItem C" type="check" toggled="false"/>
                <menuitem type="separator"/>
                <menuitem label="MenuItem D" >
```

```
                    <menuitem label="SubMenuItem D-1" type="radio" groupName="one"/>
                    <menuitem label="SubMenuItem D-2" type="radio" groupName="one" 
toggled="true"/>
                    <menuitem label="SubMenuItem D-3" type="radio" groupName="one"/>
                </menuitem>
            </xmlRoot>
        </fx:XML>
    </fx:Declarations>

    <!-- Text area to display the event information. -->
    <mx:TextArea id="ta1" width="264" height="168" x="11" y="38"/>

    <!-- Button controls to open the menus. -->
    <mx:Button id="b1" 
        label="Open XML Popup" 
        click="createAndShow();" x="10" y="10"/>
    <mx:Button id="b2" 
        label="Open Object Popup" 
        click="createAndShow2();" x="139" y="10"/>
</s:Application>
```

PopUpMenuButton control events

Because the PopUpMenuButton is a subclass of the PopUpButton control, it supports all of that control's events. When the user clicks the main button, the PopUpMenuButton control dispatches a `click` (`MouseEvent.CLICK)` event.

When the user clicks the PopUpMenuButton main button, the control dispatches an `itemClick` (MenuEvent.`ITEM_CLICK)` event that contains information about the selected menu item. Therefore, the same `itemClick` event is dispatched when the user clicks the main button or selects the current item from the pop-up menu. Because the same event is dispatched in both cases, clicking on the main button produces the same behavior as clicking on the last selected menuItem, so the main button plays the role of a frequently used menu item.

The following example shows how the PopUpMenuButton generates events and how an application can handle them:

```
<?xml version="1.0"?>
<!-- menus/PopUpMenuButtonEvents.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009" 
    xmlns:s="library://ns.adobe.com/flex/spark" 
    xmlns:mx="library://ns.adobe.com/flex/mx" 
    height="600" width="600" 
    creationComplete="initData();">
    <fx:Script>
        <![CDATA[
        import mx.events.*;
        import mx.controls.*;

        // Set the Inbox (fourth) item in the menu as the button item.
        private function initData():void {
            Menu(p1.popUp).selectedIndex=3;
        }

        // itemClick event handler, invoked when you select from the menu.
        // Shows the event's label, index properties, and the values of the 
        // label and data fields of the data provider entry specified by 
        // the event's item property.
        public function itemClickHandler(event:MenuEvent):void {
            Alert.show("itemClick event label: " + event.label
                + "  \nindex: " + event.index
                + "  \nitem.label: " + event.item.label
                + "  \nitem.data: " + event.item.data);
        }

        //Click event handler for the main button.
        public function clickHandler(event:MouseEvent):void {
            Alert.show(" Click Event currentTarget.label: "
                + event.currentTarget.label);
        }

        //The menu data provider
        [Bindable]
        public var menuDP:Array = [
            {label: "Inbox", data: "inbox"},
            {label: "Calendar", data: "calendar"},
            {label: "Sent", data: "sent"},
            {label: "Deleted Items", data: "deleted"},
            {label: "Spam", data: "spam"}
        ];
    ]]></fx:Script>
    <mx:PopUpMenuButton  id="p1"
        showRoot="true"
        dataProvider="{menuDP}"
        click="clickHandler(event);"
        itemClick="itemClickHandler(event);"/>
</s:Application>
```

When the user selects an item from the pop-up menu, the following things occur:

- The PopUpMenuButton dispatches an `itemClick` event.
- The application's `itemClickHandler()` event listener function handles the `itemClick` event and displays the information about the event in an Alert control.

When the user clicks the main button, the following things occur:

- The PopUpMenuButton control dispatches a `click` event.
- The PopUpMenuButton control dispatches an `itemClick` event.
- The application's `itemClickHandler()` event listener function handles the `itemClick` event and displays information about the selected Menu item in an Alert control.
- The application's `clickHandler()` event listener function also handles the `MouseEvent.CLICK` event, and displays the Button label in an Alert control.

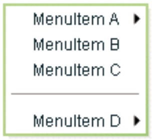
Menu control

The Menu control is a pop-up control that contains a menu of individually selectable choices. You use ActionScript to create a Menu control that pops up in response to a user action, typically as part of an event listener. Because you create a Menu control in response to an event, it does not have an MXML tag; you can create Menu controls in ActionScript only.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About the Menu Control

The following example shows a Menu control:



In this example, the MenuItem A and MenuItem D items open submenus. Submenus open when the user moves the mouse pointer over the parent item or accesses the parent item by using keyboard keys.

The default location of the Menu control is the upper-left corner of your application, at *x*, *y* coordinates 0,0. You can pass `x` and `y` arguments to the `show()` method to control the position relative to the application.

After a Menu opens, it remains visible until the user selects an enabled menu item, the user selects another component in the application, or a script closes the menu.

To create a *static* menu that stays visible all the time, use the MenuBar control or PopUpMenuButton control. For more information on the MenuBar control, see "MenuBar control" on page 1001. For more information on the PopUpMenuButton control, see "PopUpMenuButton control" on page 1003.

Create a Menu control

You cannot create a Menu control by using an MXML tag; you must create it in ActionScript.

1. Create an instance of the Menu control by calling the static ActionScript `Menu.createMenu()` method and passing the method an instance of the data provider that contains the information that populates the control as the second parameter; for example:

   ```
   var myMenu:Menu = Menu.createMenu(null, myMenuData);
   ```

   (The first parameter can optionally specify the parent container of the menu.)

   If you do not display the root node of the data provider, for example, if the data provider is an XML document in E4X format, use a third parameter with the value `false`. This parameter sets the menu's `showRoot` property. The following example creates a menu that does not show the data provider root:

   ```
   var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
   ```

   ***Note:** To hide the root node, you must set the `showRoot` property in the `createMenu` method. Setting the property after you create the menu has no effect.*

2. Display the Menu instance by calling the ActionScript `Menu.show()` method; for example:

   ```
   myMenu.show(10, 10);
   ```

***Note:** Menus displayed by using the `Menu.popUpMenu()` method are not removed automatically; you must call the `PopUpManager.removePopUp()` method on the Menu object. The `show()` method automatically adds the Menu object to the display list, and the `hide()` method automatically removes it from the display list. Clicking outside the menu (or pressing Escape) also hides the Menu object and removes it from the display list.*

Example: Creating a simple Menu control

The following example uses the `<fx:XML>` tag to define the data for the Menu control and a Button control to trigger the event that opens the Menu control:

```
<?xml version="1.0"?>
<!-- menus/SimpleMenuControl.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
    <fx:Script>
        <![CDATA[
            // Import the Menu control.
            import mx.controls.Menu;
            // Create and display the Menu control.
            private function createAndShow():void {
                var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
                myMenu.labelField="@label";
                myMenu.show(10, 10);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define the menu data. -->
        <fx:XML format="e4x" id="myMenuData">
            <root>
                <menuitem label="MenuItem A" >
                    <menuitem label="SubMenuItem A-1" enabled="false"/>
                    <menuitem label="SubMenuItem A-2"/>
                </menuitem>
                <menuitem label="MenuItem B" type="check" toggled="true"/>
                <menuitem label="MenuItem C" type="check" toggled="false"/>
                <menuitem type="separator"/>
                <menuitem label="MenuItem D" >
                    <menuitem label="SubMenuItem D-1" type="radio"
                        groupName="one"/>
                    <menuitem label="SubMenuItem D-2" type="radio"
                        groupName="one" toggled="true"/>
                    <menuitem label="SubMenuItem D-3" type="radio"
                        groupName="one"/>
                </menuitem>
            </root>
        </fx:XML>
    </fx:Declarations>
    <mx:VBox>
        <!-- Define a Button control to open the menu -->
        <mx:Button id="myButton"
            label="Open Menu"
            click="createAndShow();"/>
    </mx:VBox>
</s:Application>
```

You can assign any name to node tags in the XML data. In the previous sample, each node is named with the generic `<menuitem>` tag, but you can have used `<node>`, `<subNode>`, `<person>`, `<address>` and so on.

Because this example uses an E4X XML data source, you must specify the label field by using the E4X @ attribute specifier syntax, and you tell the control not to show the data provider root node.

Several attributes or fields, such as the `type` attribute, have meaning to the Menu control. For information on how Flex interprets and uses the data provider data, see "Specifying and using menu entry information" on page 986.

Menu control user interaction

You can use the mouse or the keyboard to interact with a Menu control. Clicking selects a menu item and closes the menu, except with the following types of menu items:

Disabled items or separators Rolling over or clicking menu items has no effect and the menu remains visible.

Submenu anchors Rolling over the items activates the submenu; clicking them has no effect; rolling onto any menu item other than one of the submenu items closes the submenu.

When a Menu control has focus, you can use the following keys to control it:

Key	Description
Down Arrow Up Arrow	Moves the selection down and up the rows of the menu. The selection loops at the top or bottom row.
Right Arrow	Opens a submenu, or moves the selection to the next menu in a menu bar.
Left Arrow	Closes a submenu and returns focus to the parent menu (if a parent menu exists), or moves the selection to the previous menu in a menu bar (if the menu bar exists).
Enter	Opens a submenu, has the effect of clicking and releasing the mouse on a row if a submenu does not exist.
Escape	Closes a menu level.

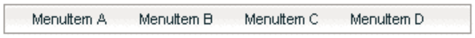
MenuBar control

A MenuBar control displays the top level of a menu as a horizontal bar of menu items, where each item on the bar can pop up a submenu. The MenuBar control interprets the data provider in the same way as the Menu control, and supports the same events as the Menu control. Unlike the Menu control, a MenuBar control is static; that is, it does not function as a pop-up menu, but is always visible in your application. Because the MenuBar is static, you can define it directly in MXML.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For more information on the Menu control, see “Menu control events” on page 990.

About the MenuBar control

The following example shows a MenuBar control:



The control shows the labels of the top level of the data provider menu. When a user selects a top-level menu item, the MenuBar control opens a submenu. The submenu stays open until the user selects another top-level menu item, selects a submenu item, or clicks outside the MenuBar area.

Creating a MenuBar control

You define a MenuBar control in MXML by using the `<mx:MenuBar>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block.

You specify the data for the MenuBar control by using the `dataProvider` property. The MenuBar control uses the same types of data providers as does the Menu control. For more information on data providers for Menu and MenuBar controls, see “Defining menu structure and data” on page 985. For more information on hierarchical data providers, see “Hierarchical data objects” on page 927.

In a simple case for creating a MenuBar control, you might use an `<fx:XML>` or `<fx:XMLList>` tag and standard XML node syntax to define the menu data provider. When you used an XML-based data provider, you must keep the following rules in mind:

- With the `<fx:XML>` tag you must have a single root node, and you set the `showRoot` property of the MenuBar control to `false`. (otherwise, your MenuBar would have only the root as a button). With the `<fx:XMLList>` tag you define a list of XML nodes, and the top level nodes define the bar buttons.
- If your data provider has a label attribute (even if it is called "label"), you must set the MenuBar control's `labelField` property and use the E4X @ notation for the label; for example:

  ```
  labelField="@label"
  ```

The `dataProvider` property is the default property of the `MenuBar` control, so you can define the XML or XMLList object as a direct child of the `<mx:MenuBar> tag`, as the following example shows:

```
<?xml version="1.0"?>
<!-- menus/MenuBarControl.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
    <!-- Define the menu; dataProvider is the default MenuBar property.
      Because this uses an XML data provider, specify the labelField and
      showRoot properties. -->
    <mx:MenuBar id="myMenuBar" labelField="@label">
        <fx:XMLList>
            <menuitem label="MenuItem A">
                <menuitem label="SubMenuItem A-1" enabled="false"/>
                <menuitem label="SubMenuItem A-2"/>
            </menuitem>
            <menuitem label="MenuItem B"/>
            <menuitem label="MenuItem C"/>
            <menuitem label="MenuItem D">
                <menuitem label="SubMenuItem D-1"
                    type="radio" groupName="one"/>
                <menuitem label="SubMenuItem D-2"
                    type="radio" groupName="one"
                    selected="true"/>
                <menuitem label="SubMenuItem D-3"
                    type="radio" groupName="one"/>
            </menuitem>
        </fx:XMLList>
    </mx:MenuBar>
</s:Application>
```

The top-level nodes in the MenuBar control correspond to the buttons on the bar. Therefore, in this example, the MenuBar control displays the four labels shown in the preceding image.

You can assign any name to node tags in the XML data. In the previous example, each node is named with the generic `<menuitem>` tag, but you can use `<node>`, `<subNode>`, `<person>`, `<address>`, and so on. Several attributes or fields, such as the `type` attribute, have meaning to the MenuBar control. For information on how Flex interprets and uses the data provider data, see "Specifying and using menu entry information" on page 986.

MenuBar control user interaction

The user interaction of the MenuBar is the same as for the Menu control, with the following difference: when the MenuBar control has the focus, the left arrow opens the previous menu. If the current menu bar item has a closed pop-up menu, the right arrow opens the current menu; if the pop-up menu is open, the right arrow opens the next menu. (The behavior wraps around the ends of the MenuBar control.)

For more information, see "Menu control user interaction" on page 1001.

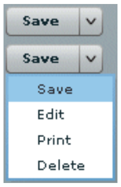
PopUpMenuButton control

The PopUpMenuButton is a PopUpButton control whose secondary button pops up a Menu control. When the user selects an item from the pop-up menu, the main button of the PopUpButton changes to show the icon and label of the selected menu item. Unlike the Menu and MenuBar controls, the PopUpMenuButton supports only a single-level menu.

For complete reference information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For more information on the Menu control, see "Menu control events" on page 990. For more information on PopUpButton controls, see "PopUpButton control" on page 702.

About the PopUpMenuButton control

The following example shows a PopUpMenuButton control before and after clicking the secondary pop-up button:



The PopUpMenuButton works as follows.

- When you click the smaller button, which by default displays a **v** icon, the control displays a pop-up menu below the button.
- When you select an item from the pop-up menu, the main PopUpMenuButton button label changes to show the selected item's label and the PopUpMenuButton control dispatches a `MenuEvent.CHANGE` event.
- When you click the main button, the PopUpMenuButton control dispatches a `MenuEvent.CHANGE` event and a `MouseEvent.ITEM_CLICK` event.

For information on handling PopUpMenuButton events, see "PopUpMenuButton control events" on page 997.

The PopUpMenuButton control lets users change the function of the main button by selecting items from the pop-up menu. The most recently selected item becomes the main button item.

This behavior is useful for buttons when there are a number of user actions, users tend to select the same option frequently, and the application developer cannot assume which option should be the default. Text editors often use such controls in their control bar for options, such as spacing, for which a user is likely to have a preferred setting, but the developer cannot determine it in advance. Microsoft Word, for example, uses such controls for specifying line spacing, borders, and text and highlight color.

You can use the PopUpButton control to create pop-up menu buttons with behaviors that differ from those of the PopUpMenuButton; for example, buttons that do not change the default action of the main button when the user selects a menu item. For more information, see "PopUpButton control" on page 702.

Creating a PopUpMenuButton control

You define a PopUpMenuButton control in MXML by using the `<mx:PopUpMenuButton>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block.

You specify the data for the PopUpMenuButton control by using the `dataProvider` property. For information on valid data providers, including their structure and contents, see "Defining menu structure and data" on page 985.

By default, the initially selected item is the first item in the pop-up menu dataProvider, and the default main button label is the item's label, as determined by the `labelField` or `labelFunction` property. To set the initial main button label to a specific item's label and functionality, write a listener for the PopUpMenuButton control's `creationComplete` event that sets the `selectedIndex` property of the Menu subcontrol, as follows:

```
Menu(MyPopUpControl.popUp).selectedIndex=2;
```

You must cast the PopUpMenuButton control's `popUp` property to a Menu because the property type is IUIComponent, not Menu.

You can also use the `label` property of the PopUpMenuButton control to set the main button label.

When a popped up menu closes, it loses its selection and related properties.

Note: *You must use the PopUpMenuButton's `creationComplete` event, not the `initialize` event, to set the main button label from the data provider.*

Example: Creating a PopUpMenuButton control

The following example creates a PopUpMenuButton control by using an E4X XML data provider.

```
<?xml version="1.0"?>
<!-- menus/PopUpMenuButtonControl.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Menu

            // The initData function sets the initial value of the button
            // label by setting the Menu subcontrol's selectedIndex property.
            // You must cast the popUp property to a Menu.
            private function initData():void {
                Menu(pb2.popUp).selectedIndex=2;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:XML format="e4x" id="dp2">
            <root>
                <editItem label="Cut"/>
                <editItem label="Copy"/>
                <editItem label="Paste"/>
                <separator type="separator"/>
                <editItem label="Delete"/>
            </root>
        </fx:XML>
    </fx:Declarations>
    <mx:PopUpMenuButton id="pb2"
        dataProvider="{dp2}"
        labelField="@label"
        showRoot="false"
        creationComplete="initData();"/>
</s:Application>
```

Because this example uses an E4X XML data source, you must specify the label field by using the E4X @ attribute specifier syntax, and you must tell the control not to show the data provider root node.

Using the label property

The `label` property of the PopUpMenuButton control specifies the contents of the label on the main button, and overrides any label from the pop-up menu that is determined by the `labelField` or `labelFunction` property. The `label` property is useful for creating a main button label with fixed and a variable parts; for example, a mail "Send to:" button where only the destination text is controlled by the pop-up menu, so the main button could say "Send to: Inbox" or "Send to: Trash" based on the selection from a menu that lists "Menu" and "Trash."

To use a dynamic `label` property, use a PopUpMenuButton control `change` event listener to set the label based on the event's `label` property, as in the following example:

```
<?xml version="1.0"?>
<!-- menus/PopUpMenuButtonLabel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.events.MenuEvent;

            public function itemClickHandler(event:MenuEvent):void {
                event.currentTarget.label= "Send to: " + event.label;
            }
            [Bindable]
            public var menuData:Array = [
                {label: "Inbox", data: "inbox"},
                {label: "Calendar", data: "calendar"},
                {label: "Sent", data: "sent"},
                {label: "Deleted Items", data: "deleted"},
                {label: "Spam", data: "spam"}
            ];
        ]]>
    </fx:Script>
    <mx:PopUpMenuButton id="p1"
        showRoot="true"
        dataProvider="{menuData}"
        label="Send to: Inbox"
        itemClick="itemClickHandler(event);"/>
</s:Application>
```

PopUpMenuButton user interaction

The user interaction of the PopUpMenuButton control main button and secondary button is the same as for the PopUpButton control. The user interaction with the pop-up menu is the same as for the Menu control. For more information on the PopUpButton user interaction, see “User interaction” on page 704. For more information on Menu control user interaction, see “Menu control user interaction” on page 1001.

MX item renderers and item editors

You can customize the appearance and behavior of cells in MX list-based data provider controls that you use in applications built with Adobe® Flex®, including the DataGrid, HorizontalList, List, Menu, MenuBar, TileList, and Tree controls. To control the appearance of cells in list controls, you create custom item renderers and item editors.

For more information about creating more advanced item editors, see “Advanced data display with MX item editors” on page 1048.

For information on using item renderers with Spark components, see “Custom Spark item renderers” on page 470.

About MX item renderers

MX supports several controls that you can use to represent lists of items. These controls let the application user scroll through the item list and select one or more items from the list. All MX list components are derived from the ListBase class, and include the following controls:

- DataGrid

- HorizontalList
- List
- Menu
- MenuBar
- TileList
- Tree

A list control gets its data from a data provider. A *data provider* is a collection that contains a data object such as an Array or XMLList object. For example, a Tree control reads data from a data provider to define the structure of the tree and any associated data that is assigned to each tree node. *Collections* are objects that contain a set of methods that let you access, sort, filter, and modify the data items in a data object. The standard collection types are the ArrayCollection and XMLListCollection classes, for working with Array-based and XMLList-based data, respectively.

Collections provide a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same collection, switch the collection that a component uses as a data provider at run time, or modify a collection so that changes are reflected by all components that use it as a data provider.

You can think of the data provider as the model, and the Flex components as the view of the model. By separating the model from the view, you can change one without changing the other. Each list control has a default mechanism for controlling the display of data, or view, and lets you override that default. To override the default view, you create a custom *item renderer*.

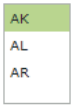
In addition to controlling the display of data using an item renderer, the DataGrid, List, and Tree controls let users edit the data. For example, you could let users update the Quantity column of a DataGrid control if you are using it for a shopping cart. As with the display of data, the list controls have a default mechanism for letting users edit data that you can override by creating a custom *item editor*.

Default item rendering and cell editing in MX

Each MX list-based control has a default item renderer defined for it. The simplest item renderer is the DataGridItemRenderer class, which defines the item renderer for the DataGrid control. This item renderer assumes that each element in a data provider is a text string.

Other item renderers include the ListItemRenderer, MenuItemRenderer, MenuBarItem, TileListItemRenderer, and TreeItemRenderer. By default, these item renderers combine an image with text.

For example, the following image shows a List control that displays three items using its default item renderer:



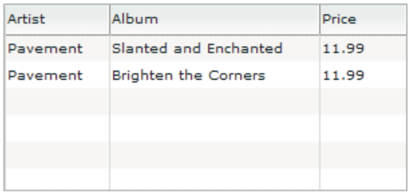
AK
AL
AR

In this example, the List control uses the default item renderer to display each of the three strings that represent postal codes for Alaska, Alabama, and Arkansas. You use the following code to create this List control:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\ListDefaultRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:List width="50" height ="75">
        <mx:dataProvider>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
            <fx:String>AR</fx:String>
        </mx:dataProvider>
    </mx:List>
</s:Application>
```

Because the data in this example is inline static data, it is not necessary to explicitly wrap it in an ArrayCollection instance. However, when you work with data that could change, it is always best to specify a collection explicitly; for more information, see “Data providers and collections” on page 898.

In the next example, the data provider for the DataGrid control contains fields for an artist, album name, and price. Each of these fields appears in the DataGrid control using the default item renderer, which displays data as text.

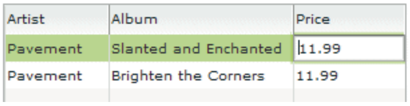


Artist	Album	Price
Pavement	Slanted and Enchanted	11.99
Pavement	Brighten the Corners	11.99

The following code creates the example shown in the previous image:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\DGDefaultRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >

    <fx:Script>
        <![CDATA[
          import mx.collections.ArrayCollection;

          [Bindable]
          private var initDG:ArrayCollection = new ArrayCollection([
            {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
            {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}
          ]);
        ]]>
    </fx:Script>
    <mx:Panel paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10">
        <mx:DataGrid id="myGrid" dataProvider="{initDG}"
            width="100%" editable="true">
            <mx:columns>
                <mx:DataGridColumn dataField="Artist" resizable="true"/>
                <mx:DataGridColumn dataField="Album" resizable="true"/>
                <mx:DataGridColumn dataField="Price" resizable="true"/>
            </mx:columns>
        </mx:DataGrid>
    </mx:Panel>
</s:Application>
```

An editable component lets the user modify the data in the list control, and propagates the changes in the data back to the underlying data provider of the control. The DataGrid, List, and Tree controls have a property named `editable` that, if set to `true`, lets you edit the contents of a cell. By default, the list controls use a TextInput control as the item editor. That means when you select a cell in the list control, Flex opens a TextInput control that contains the current contents of the cell and lets you edit it, as the following image shows:



Artist	Album	Price
Pavement	Slanted and Enchanted	11.99
Pavement	Brighten the Corners	11.99

In this image, you use the default item editor to edit the price of the first album in the DataGrid control.

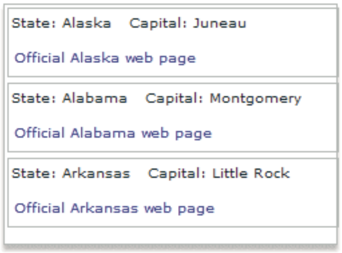
For more information on item editors, see “Advanced data display with MX item editors” on page 1048.

Using custom MX item renderers and item editors

To control the display of a list component, you write a custom item renderer or custom item editor. Your custom item renderers and item editors still use the underlying functionality of the list control, but let you control the display and editing of the data. Custom item renderers and item editors provide you with several advantages:

- You can create a more compelling user interface by replacing the display of text with a more user-intuitive appearance.
- You can combine multiple elements in a single list item, such as a label and an image.
- You can programmatically control the display of the data.

The following List control is a modification of the List control in the section “Default item rendering and cell editing in MX” on page 1007. In this example, you use a custom item renderer to display the state name, capital, and a URL to the state’s official web site in each list item:



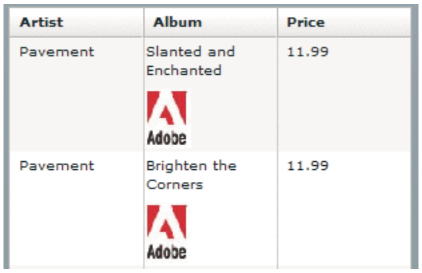
State: Alaska Capital: Juneau
Official Alaska web page

State: Alabama Capital: Montgomery
Official Alabama web page

State: Arkansas Capital: Little Rock
Official Arkansas web page

For the code for this example, see “Example: Using an item renderer with an MX List control” on page 1043.

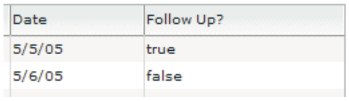
In the next image, you use a default item renderer for the first and third columns of the DataGrid control. You use a custom item renderer for the second column to display the album cover along with the album title in the DataGrid control.



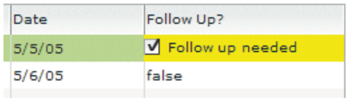
Artist	Album	Price
Pavement	Slanted and Enchanted	11.99
Pavement	Brighten the Corners	11.99

For the code for this example, see “Creating a complex inline item renderer or item editor” on page 1027.

Just as you can define a custom item renderer to control the display of a cell, you can use a custom item editor to edit the contents of the cell. For example, if the custom item renderer displays an image, you could define a custom item editor that uses a ComboBox control that lets users select an image from a list of available images. Or you could use a CheckBox control to let the user set a `true` or `false` value for a cell, as the right side of the following example shows:



Date	Follow Up?
5/5/05	true
5/6/05	false



Date	Follow Up?
5/5/05	☑ Follow up needed
5/6/05	false

For the code for this example, see “Creating a simple item editor component” on page 1035.

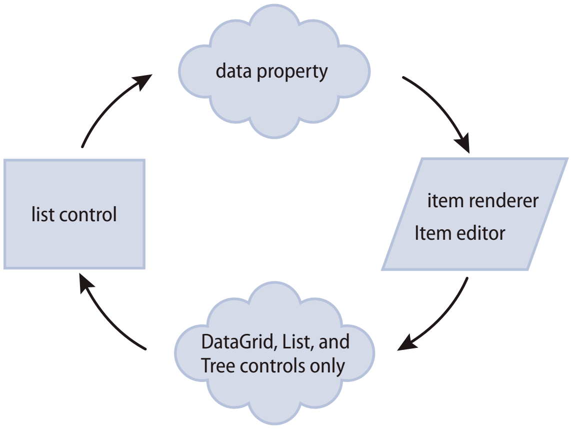
Using an item renderer does not imply that the control also has an item editor. Often, you do not allow your controls to be edited; that is, they are for display only.

You can also use an item editor without a corresponding item renderer. For example, you could display information such as a male/female or true/false as text using the default item renderer. But, you could then use a custom item editor with a ComboBox control that provides the user only a limited set of options to enter into the cell when changing the value.

MX item renderer and item editor architecture

In order for an item renderer or item editor to work with the contents of a list control, the list control must be able to pass information to the item renderer or item editor. An item editor must also be able to pass updated information back to the list control.

The following image shows the relationship of a list control to an item renderer or item editor:



To pass information to an item renderer or item editor, Flex sets the `data` property of the item renderer or item editor to the cell data. In the previous image, you can see the `data` property that is used to pass the data to the item renderer or item editor.

By default, an item editor can pass back a single value to the list control that becomes the new value of the edited cell. To return more than one value, you must write additional code to return the data back to the list control.

Any Flex component that you want to use in an item renderer or item editor, and that requires access to the cell data of the list control, must implement the IDataRenderer interface to support the `data` property. You can use additional components in an item renderer that do not support the `data` property, if those components do not need to access data passed from the list control.

The contents of the `data` property depend on the type of control, as the following table shows:

Control	Contents of the data property
DataGrid	Contains the data provider element for the entire row of the DataGrid control. That means if you use a different item renderer for each cell of the DataGrid control, each cell in a row receives the same information in the `data` property.
Tree List	Contains the data provider element for the node of the List or Tree control.
HorizontalList TileList	Contains the data provider element for the cell.
Menu	Contains the data provider element for the menu item.
MenuBar	Contains the data provider element for the menu bar item.

For more information about item editors, see “Advanced data display with MX item editors” on page 1048.

About MX item renderer and item editor interfaces

The Flex item renderer and item editor architecture is defined by several interfaces. Flex components that you can use as item renderers and item editors implement one or more of these interfaces. If you create your own custom component to use as an item renderer or item editor, your component must implement one or more of these interfaces, depending on how you intend to use it.

The following table describes the interfaces that you use with item renderers and item editors:

Interface	Use
IDataRenderer	Defines the `data` property used to pass information to an item renderer or item editor. All item renderers and item editors must implement this interface. Many Flex components implement this interface, such as the chart renderer classes and many Flex controls.
IDropInListItemRenderer	Defines the `listData` property required by drop-in item renderers and item editors. All drop-in item renderers and item editors must implement this interface. For more information, see "Creating MX drop-in item renderers and item editors" on page 1022. The `listData` property is of type BaseListData, where the BaseListData class has three subclasses: DataGridListData, ListData, TreeListData. The actual data type of the value of the `listData` property depends on the control using the drop-in item renderer or item editor. For a DataGrid control, the value is of type DataGridListData; for a List control, the value is of type ListData; and for a Tree control, the value is of type TreeListData.
IListItemRenderer	Defines the complete set of interfaces that an item renderer or item editor must implement to be used with any Flex control other than a Chart control. A renderer for a Chart control has to implement only the IDataRenderer interface. The set of interfaces includes the following: IDataRenderer, IFlexDisplayObject, ILayoutClient, IStyleable, and IUIComponent. The UIComponent class implements all of these interfaces, except the IDataRenderer interface. Therefore, if you create a custom item renderer or item editor as a subclass of the UIComponent class, you have to implement only the IDataRenderer interface. If you implement a drop-in item renderer or item editor as a subclass of the UIComponent class, you have to implement only the IDataRenderer and IDropInListItemRenderer interfaces.

Application layout with item renderers and item editors

All list controls, with the exception of the TileList and HorizontalList controls, ignore item renderers when calculating the default height and width of their contents. By default, item renderers are sized to the full width of a control, or the width of the column for a DataGrid control. Therefore, if you define a custom item renderer that requires a size other than its default, you should explicitly size the control.

Vertical controls such as the List and Tree controls use the `rowHeight` property to determine the default cell height. You can also set the `variableRowHeight` property to `true` if each row has a different height. Otherwise, Flex sets the height of each cell to 20 pixels.

For the HorizontalList control, Flex can use the value of the `columnWidth` property, if specified. For the TileList control, Flex uses the values of the `columnWidth` and `rowHeight` properties. If you omit these properties, Flex examines the control's data provider to determine the default size of each cell, including an item renderer, if specified. Otherwise, Flex sets the height and width of each cell to 50 pixels.

Creating an MX item renderer and item editor

One of the first decisions that you must make when using custom item renderers and item editors is how to implement them. Flex lets you define item renderers and item editors in three different ways:

Drop-in item renderers and item editors Insert a single component to define an item renderer or item editor. For more information, see "Using an MX drop-in item renderer or item editor" on page 1013.

Inline item renderers and item editors Define one or more components using child tags of the list control to define an item renderer or item editor. For more information, see "Using an MX inline item renderer or item editor" on page 1014.

Item renderer and item editor components Define an item renderer or item editor as a reusable component. For more information, see "Using a component as an MX item renderer or item editor" on page 1015.

The following sections describe each technique.

Using an MX drop-in item renderer or item editor

For simple item renderers and item editors, such as using a NumericStepper control to edit a field of a DataGrid control, you can use a drop-in item editor. A *drop-in item renderer* or *drop-in item editor* is a Flex component that you specify as the value of the `itemRenderer` or `itemEditor` property of a list control.

In the following example, you specify the NumericStepper control as the item editor for a column of the DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\dropin\DropInNumStepper.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1" headerText="Order #"/>
            <mx:DataGridColumn dataField="quant"
                headerText="Qty"
                itemEditor="mx.controls.NumericStepper"
                editorDataField="value"
            />
        </mx:columns >
    </mx:DataGrid>
</s:Application>
```

In this example, when the user selects a cell of the column to edit it, Flex displays the NumericStepper control in that cell. Flex automatically uses the `data` property to populate the NumericStepper control with the current value of the column.

You use the `editorDataField` property to specify the property of the item editor that returns the new data for the cell. By default, the list control uses the `text` field of the TextInput control to supply the new value; therefore, the default value of the `editorDataField` property is `"text"`. In this example, you specify that the `value` field of the NumericStepper supplies the new value for the cell.

You can use only a subset of the Flex components as drop-in item renderers and item editors—those components that implement the IDropInListItemRenderer interface. For more information on using drop-in item renderers and item editors, and for a list of controls that support drop-in item renderers and item editors, see "Creating MX drop-in item renderers and item editors" on page 1022.

Using an MX inline item renderer or item editor

In the section "Using an MX drop-in item renderer or item editor" on page 1013, the example shows how easy it is to use drop-in item renderers and item editors. The only drawback to using them is that you cannot configure them. You can specify the drop-in item renderers and item editors only as the values of a list control property.

To create more flexible item renderers and item editors, you develop your item renderer or item editor as an inline component. In the next example, you modify the previous example to use a NumericStepper control as an inline item editor. With an inline item editor, you can configure the NumericStepper control just as if you used it as a stand-alone control.

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineNumStepper.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1" headerText="Order #"/>
            <mx:DataGridColumn dataField="quant" editorDataField="value" headerText="Qty">
                <mx:itemEditor>
                    <fx:Component>
                        <mx:NumericStepper stepSize="1" maximum="50"/>
                    </fx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you define the NumericStepper control as the item editor for the column, and specify a maximum value of 50 and a step size of 1 for the NumericStepper control.

For more information on creating inline item renderers and item editors, see “Creating MX inline item renderers and editors” on page 1025.

Using a component as an MX item renderer or item editor

One disadvantage of using drop-in and inline item renderers and editors is that you define the item renderer or editor in the application file; therefore, it is not reusable in another location in the application, or in another application. You can create a reusable item renderer or item editor as a Flex component, and then use that component anywhere in an application that requires the item renderer.

For example, the following code uses a NumericStepper control to define a custom item editor as an MXML component:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\myComponents\NSEditor.mxml -->
<mx:NumericStepper xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    stepSize="1"
    maximum="50"/>
```

The custom MXML component defines the item editor as a NumericStepper control. In this example, the custom item editor is named NSEditor and is implemented as an MXML component in the NSEditor.mxml file. You place the file NSEditor.mxml in the myComponents directory beneath your main application directory.

You can then use this component anywhere in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\MainNSEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG" dataProvider="{myDP}"
            variableRowHeight="true"
            editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1"
                headerText="Order #"/>
            <mx:DataGridColumn dataField="quant"
                itemEditor="myComponents.NSEditor"
                editorDataField="value"/>
        </mx:columns >
    </mx:DataGrid>
</s:Application>
```

When the user selects a cell in the quant column of the DataGrid control, Flex displays a NumericStepper control with the current cell value.

You might have several locations in your application where users can modify a numeric value. You defined this item editor as a custom component; therefore, you can reuse it anywhere in your application.

For more information on creating item renderers and item editors as components, see “Creating MX item renderers and item editor components” on page 1033.

Using editable controls in an MX item renderer

Item renderers do not impose restrictions on the types of Flex components that you can use in them. For example, you can use controls, such as the Label, LinkButton, Button, and Text controls, to display data, but these controls do not let the user modify the contents of the control.

Or you can use controls such as the CheckBox, ComboBox, and TextInput controls that both display data and let users interact with the control to modify or change it. For example, you could use a CheckBox control to display a selected (`true` value) or unselected (`false` value) cell in a DataGrid control.

When the user selects the cell of the DataGrid control that contains the CheckBox control, the user can interact with the control to change its state. To the user, it appears that the DataGrid control is editable.

However, an item renderer by default is not connected to the editing mechanism of the list control; it does not propagate changes to the list control's data provider, nor does it dispatch an event when the user modifies the cell. Although the list control appears to the user to be editable, it really is not.

In another example, the user changes the value of the CheckBox control, and then sorts the DataGrid column. But the DataGrid sorts the cell by the value in the data provider, not the value in the CheckBox control, so the user perceives that the sort works incorrectly.

You can manage this situation in several ways, including the following:

- In your item renderer, do not use controls that let users modify them (CheckBox, ComboBox, and others).
- Create custom versions of these controls to prohibit user interaction with them.
- Use the `rendererIsEditor` property of the list control to specify that the item renderer is also an item editor. For more information, see "Example: Using an item renderer as an item editor" on page 1066.
- Write your own code for the item renderer and hosting control to pass data back from the item renderer when you do let users interact with it.

Setting the itemRenderer or itemEditor property in ActionScript

The `itemRenderer` and `itemEditor` properties are of type IFactory. When you set these properties in MXML, the MXML compiler automatically casts the property value to the type ClassFactory, a class that implements the IFactory interface.

When you set these properties in ActionScript, you must explicitly cast the property value to ClassFactory, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\AppListStateRendererEditorAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.core.ClassFactory;

            // Cast the value of the itemRenderer property
            // to ClassFactory.
            public function initCellEditor():void {
                myList.itemRenderer=new ClassFactory(RendererState);
            }
        ]]>
    </fx:Script>
    <mx:List id="myList" variableRowHeight="true"
            height="180" width="250"
            initialize="initCellEditor();">
        <mx:dataProvider>
            <fx:Object label="Alaska"
                data="Juneau"
                webPage="http://www.state.ak.us/"/>
            <fx:Object label="Alabama"
                data="Montgomery"
                webPage="http://www.alabama.gov/" />
            <fx:Object label="Arkansas"
                data="Little Rock"
                webPage="http://www.state.ar.us/"/>
        </mx:dataProvider>
    </mx:List>
</s:Application>
```

Shown below is the code for the RenderState.mxml item renderer:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\RendererState.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            // Import Event and URLRequest classes.
            import flash.events.Event;
            import flash.net.URLRequest;

            private var u:URLRequest;

            // Event handler to open URL using navigateToURL().
            private function handleClick(eventObj:Event):void {
                u = new URLRequest(data.webPage);
                navigateToURL(u);
            }
        ]]>
    </fx:Script>

    <mx:HBox >
        <!-- Use Label controls to display state and capital names. -->
        <mx:Label id="State" text="State: {data.label}"/>
        <mx:Label id="Statecapital" text="Capital: {data.data}" />
    </mx:HBox>
    <!-- Define the Link control to open a URL. -->
    <mx:LinkButton id="webPage" label="Official {data.label} web page"
        click="handleClick(event);" color="blue"  />
</mx:VBox>
```

About the MX item renderer and item editor life cycle

Flex creates instances of item renderers and item editors as needed by your application for display and measurement purposes. Therefore, the number of instances of an item renderer or item editor in your application is not necessarily directly related to the number of visible item renderers. Also, if the list control is scrolled or resized, a single item renderer instance may be reused to represent more than one data item. Thus, you should not make assumptions about how many instances of your item renderer and item editor are active at any time.

Because Flex can reuse an item renderer, ensure that you fully define its state. For example, you use a CheckBox control in an item renderer to display a `true` (checked) or `false` (unchecked) value based on the current value of the `data` property. A common mistake is to assume that the CheckBox control in the item renderer is always in its default state of unchecked. Developers then write an item renderer to inspect the `data` property for a value of `true` and set the CheckBox control to checked if found.

However, you must take into account that the CheckBox may already be checked. So, you must inspect the `data` property for a value of `false`, and explicitly uncheck the control if it is checked.

Accessing the listData property

If a component implements the IDropInListItemRenderer interface, you can use its `listData` property to obtain information about the data passed to the component when you use the component in an item renderer or item editor. The `listData` property is of type BaseListData, where the BaseListData class defines the following properties:

Property	Description
owner	A reference to the list control that uses the item renderer or item editor.
rowIndex	The index of the row of the DataGrid, List, or Tree control relative to the currently visible rows of the control, where the first row is at an index of 1.
label	The text representation of the item data based on the List class's itemToLabel() method.

The BaseListData class has three subclasses: DataGridListData, ListData, TreeListData that define additional properties. For example, the DataGridListData class adds the columnIndex and dataField properties that you can access from an item renderer or item editor.

The data type of the value of the listData property depends on the control that uses the item renderer or item editor. For a DataGrid control, the value is of type DataGridListData; for a List control, the value is of type ListData; and for a Tree control, the value is of type TreeListData.

The TextArea control is one of the Flex controls that implements the IDropInListItemRenderer interface The item renderer in the following example uses the listData property of the TextArea control to display the row and column of each item renderer in the DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\RendererDGListData.mxml -->
<mx:TextArea xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    preinitialize="initTA();">
    <fx:Script>
        <![CDATA[

            import mx.controls.dataGridClasses.DataGridListData;
            import flash.events.Event;

            public function initTA():void {
                addEventListener("dataChange", handleDataChanged);
            }
            public function handleDataChanged(event:Event):void {
                // Cast listData to DataGridListData.
                var myListData:DataGridListData =
                    DataGridListData(listData);

                // Access information about the data passed
                // to the cell renderer.
                text="row index: " + String(myListData.rowIndex) +
                    " column index: " + String(myListData.columnIndex);
            }
        ]]>
    </fx:Script>
</mx:TextArea>
```

Because you use this item renderer in a DataGrid control, the data type of the value of the listData property is DataGridListData. You use the dataChange event in this example to set the contents of the TextArea control every time the data property changes.

The DataGrid control in the following example uses this item renderer to display the column and row index of each cell of the DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainDGListDataRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var initDG:ArrayCollection = new ArrayCollection([
                { Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', Date: '5/5/05'},
                { Company: 'Allied', Contact: 'Jane Smith',
                    Phone: '617-555-3434', Date: '5/6/05'}
            ]);
        ]]>
    </fx:Script>
    <mx:Panel paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10" >
        <mx:DataGrid id="myGrid" dataProvider="{initDG}"
                variableRowHeight="true">
            <mx:columns>
                <mx:DataGridColumn dataField="Company"
                    itemRenderer="myComponents.RendererDGListData"/>
                <mx:DataGridColumn dataField="Contact"
                    itemRenderer="myComponents.RendererDGListData"/>
                <mx:DataGridColumn dataField="Phone"
                    itemRenderer="myComponents.RendererDGListData"/>
                <mx:DataGridColumn dataField="Date"
                    itemRenderer="myComponents.RendererDGListData"/>
            </mx:columns>
        </mx:DataGrid>
    </mx:Panel>
</s:Application>
```

Handling data binding warnings from the compiler

Many of the following examples, and those in “Advanced data display with MX item editors” on page 1048, define the application data as an ArrayCollection of Objects. However, you might get compiler warning for these examples because the Object class does not support data binding. That does not matter with these examples because the data is static.

If your data is dynamic, and you want it to support data binding, you can instead define your own data class that supports binding. For example, you could create the following subclass of Object that supports data binding, and use it instead:

```
package
{
    [Bindable]
    public class MyBindableObj extends Object
    {
        public function MyBindableObj()  {
            super();
        }

        public var Artist:String = new String();

        public var Album:String = new String();

        public var Price:Number = new Number();

        public var Cover:String = new String();
    }
}
```

By inserting the `[Bindable]` metadata tag before the class definition, you specify that all public properties support data binding. For more information on data binding, see “Data binding” on page 299.

Creating MX drop-in item renderers and item editors

Several MX controls are designed to work as item renderers and item editors. This lets you specify these controls as values of the `itemRenderer` or `itemEditor` property. When you specify one of these controls as a property value, it is called a drop-in item renderer or drop-in item editor.

To use a component as a drop-in item renderer or drop-in item editor, a component must implement the IDropInListItemRenderer interface. The following controls implement the IDropInListItemRenderer interface, making them usable directly as a drop-in item renderer or drop-in item editor:

- Button
- CheckBox
- DateField
- Image
- Label
- NumericStepper
- Text
- TextArea
- TextInput

You can define your own components for use as drop-in item renderers or drop-in item editors. The only requirement is that they, too, implement the IDropInListItemRenderer interface.

Using MX drop-in item renderers and item editors

When you use a control as a drop-in item renderer, Flex sets the control’s default property to the current value of the cell. When you use a control as an item editor, the initial value of the cell becomes the current value of the control. Any edits made to the cell are copied back to the data provider of the list control.

The following table lists the MX components that you can use as drop-in item renderers and item editors, and the default property of the component:

Control	Default property	Notes
Button	`selected`	Cannot specify a label for the Button control.
CheckBox	`selected`	If used as a drop-in item renderer in a Tree control, the Tree displays only check boxes with no text.
DateField	`selectedDate`	Requires that the data provider of the list control has a field of type Date.
Image	`source`	Set an explicit row height by using the `rowHeight` property, or set the `variableRowHeight` property to `true` to size the row correctly.
Label	`text`	
NumericStepper	`value`	
Text	`text`	
TextArea	`text`	
TextInput	`text`	

In the following example, you use the NumericStepper, DateField, and CheckBox controls as the drop-in item renderers and item editors for a DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\CBInlineCellEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", contact:"John Doe",
                    quant:3, solddate:new Date(2005, 0, 1), Sent:true},
                {label1:"Order #2315", contact:"Jane Doe",
                    quant:3, solddate:new Date(2005, 0, 5), Sent:false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG"
            dataProvider="{myDP}"
            variableRowHeight="true"
            width="500" height="250"
            editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="label1"
                headerText="Order #"
                editable="false"/>
            <mx:DataGridColumn dataField="quant"
                headerText="Quantity"
                itemEditor="mx.controls.NumericStepper"
                editorDataField="value"/>
            <mx:DataGridColumn dataField="solddate"
                headerText="Date"
                itemRenderer="mx.controls.DateField"
                rendererIsEditor="true"
                editorDataField="selectedDate"/>
            <mx:DataGridColumn dataField="Sent"
                itemRenderer="mx.controls.CheckBox"
                rendererIsEditor="true"
                editorDataField="selected"/>
        </mx:columns >
    </mx:DataGrid>
</s:Application>
```

To determine the field of the data provider used to populate the drop-in item renderer, Flex uses the value of the `dataField` property for the `<mx:DataGridColumn>` tag. In this example, you do the following:

- You set the `dataField` property of the second column to `quant`, and define the NumericStepper control as the item editor. Therefore, the column displays the cell value as text, and opens the NumericStepper control when you select the cell to edit it. The NumericStepper control displays the current value of the cell, and any changes you make to it are copied back to the data provider of the DataGrid control.
- You set the `dataField` property of the third column to `solddate`, and define the DateField control as the item renderer and item editor by setting the `rendererIsEditor` property to `true`. The data provider defines the data as an object of type Date, which is required to use the DateField control as the item renderer or item editor. Therefore, the column displays the date value using the DateField control, and also uses the DateField control to edit the cell value.
- You set the `dataField` property of the fourth column to `Sent`, and define the CheckBox control as the item renderer and item editor. Typically, you use `itemEditor` property to specify a different class as the item editor, and specify to use the same control as both the item renderer and item editor by setting the `rendererIsEditor` property to `true`. The CheckBox control displays a check mark in the cell if the Sent field for the row is set to `true`, and an empty check box if the field is set to `false`.

  For more information on using an item renderer as an item editor, see "Creating an editable cell in MX" on page 1049.

***Note:** When you use a CheckBox control as a drop-in item renderer, the control appears flush against the left cell border. To center a CheckBox control, or any drop-in item renderer, create a custom item renderer that wraps the control in a container, such as the HBox container. However, the addition of the container can affect application performance when you are rendering large amounts of data. For more information, see "Creating MX item renderers and item editor components" on page 1033.*

When you use the Image control as a drop-in item renderer, you usually have to set the row height to accommodate the image, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\DGDropInImageRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover:'../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover:'../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" dataProvider="{initDG}" rowHeight="50">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover"
                itemRenderer="mx.controls.Image"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you use the Image control to display the album cover in a cell of the DataGrid control.

Requirements of a drop-in item renderer in a List control

When you use the `itemRenderer` property of the List control to specify a drop-in item renderer, the data in the data provider must be of the type expected by the item renderer control. For example, if you use an Image control as a drop-in item renderer, the data provider for the List control must have String data in the field.

Creating MX inline item renderers and editors

You define inline item renderers and item editors in the MXML definition of a component. Inline item renderers and item editors give you complete control over item rendering and cell editing.

Creating a simple inline item renderer or item editor

A simple inline item renderer contains a single control that supports the `data` property. Flex automatically copies the current cell data to the item renderer, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\InlineImageRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: '../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: '../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" dataProvider="{initDG}" rowHeight="50">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover">
                <mx:itemRenderer>
                    <fx:Component>
                        <mx:Image height="45"/>
                    </fx:Component>
                </mx:itemRenderer>
            </mx:DataGridColumn>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you use an Image control to display the album cover.

A simple inline item editor contains a single control that supports the `data` property. Flex automatically copies the current cell data to the item renderer or item editor, and copies the new cell data back to the list control based on the value of the `editorDataField` property, as the following example item editor shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineNumStepper.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1" headerText="Order #"/>
            <mx:DataGridColumn dataField="quant" editorDataField="value" headerText="Qty">
                <mx:itemEditor>
                    <fx:Component>
                        <mx:NumericStepper stepSize="1" maximum="50"/>
                    </fx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you return the new cell value by using the value property of the NumericStepper control.

Creating a complex inline item renderer or item editor

A complex item renderer or item editor defines multiple controls. For example, the section "Default item rendering and cell editing in MX" on page 1007 showed a DataGrid control that displayed information about albums by using three text fields. You could add a visual element to your DataGrid control to make it more compelling. To do that, you modify the data provider so that it contains a reference to a JPEG image of the album cover.

Rather than displaying the album name and album image in separate cells of the DataGrid control, you can use an inline item renderer to make them appear in a single cell, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\InlineDGRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[

            import mx.collections.ArrayCollection;

            // Variable in the Application scope.
            public var localVar:String="Application localVar";

            // Data includes URL to album cover.
            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover:'../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover:'../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" dataProvider="{initDG}"
            variableRowHeight="true">
        <mx:columns>
                <mx:DataGridColumn dataField="Artist"/>
                <mx:DataGridColumn dataField="Album">
                    <mx:itemRenderer>
                        <fx:Component>
                            <mx:VBox>
                                <mx:Text id="albumName"
                                    width="100%" text="{data.Album}"/>
                                <mx:Image id="albumImage"
                                    height="45" source="{data.Cover}"/>
                            </mx:VBox>
                        </fx:Component>
                    </mx:itemRenderer>
                </mx:DataGridColumn>
                <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In the preceding example, you define three columns in the DataGrid control, and assign your item renderer to the second column. For an image that shows the output of this application, see “Using custom MX item renderers and item editors” on page 1009.

Notice that the Text and Image controls in the item renderer both use the `data` property to initialize their values. This is necessary because you defined multiple controls in the item renderer, and Flex cannot automatically determine which data element in the data provider is associated with each control of the item renderer.

Even if the top-level container of an inline item renderer has a single child control, you must use the `data` property to initialize the child control, as the following example shows:

```
<mx:DataGridColumn dataField="Cover">
    <mx:itemRenderer>
        <fx:Component>
            <mx:VBox horizontalAlign="center">
                <mx:Image height="45" source="{data.Cover}"/>
            </mx:VBox>
        </fx:Component>
    </mx:itemRenderer>
</mx:DataGridColumn>
```

In the preceding example, you make the Image control a child of a VBox container so that you can align the image in the cell. Because the Image control is now a child of the VBox container, you must initialize it by using the `data` property.

You can define multiple controls in a complex inline item editor, which lets you edit multiple values of the data provider for the list control. Alternatively, you can define a complex inline item editor so it returns a value other than a String. For more information, see “Advanced data display with MX item editors” on page 1048.

Items allowed in an inline component

The only restriction on what you can and cannot do in an inline item renderer or editor is that you cannot create an empty `<fx:Component></fx:Component>` tag. For example, you can combine effect and style definitions in an inline item renderer or editor along with your rendering and editing logic.

You can include the following items in an inline item renderer or editor:

- Binding tags
- Effect tags
- Metadata tags
- Model tags
- Scripts tags
- Service tags
- State tags
- Styles tags
- XML tags
- `id` attributes, except for the top-most component

Using the Component tag

You use the `<fx:Component>` tag to define an inline item renderer or item editor in an MXML file.

Defining the scope in an Component tag

The `<fx:Component>` tag defines a new scope in an MXML file, where the local scope of the item renderer or item editor is defined by the MXML code block delimited by the `<fx:Component>` and `</fx:Component>` tags. To access elements outside of the local scope of the item renderer or item editor, you prefix the element name with the `outerDocument` keyword.

For example, you define one variable named localVar in the scope of the main application, and another variable with the same name in the scope of the item renderer. From within the item renderer, you access the application’s localVar by prefixing it with `outerDocument` keyword, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\InlineDGImageScope.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Variable in the Application scope.
            [Bindable]
            public var localVar:String="Application localVar";

            // Data includes URL to album cover.
            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                { Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover:'../assets/slanted.jpg'},
                { Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover:'../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid" dataProvider="{initDG}" width="100%"
            variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover">
                <mx:itemRenderer>
                  <fx:Component>
                    <mx:VBox>
                        <fx:Script>
                          <![CDATA[
                            // Variable in the renderer scope.
                            [Bindable]
                            public var localVar:String="Renderer localVar";
                          ]]>
                         </fx:Script>
                         <mx:Text id="albumName"
                            width="100%"
                            selectable="false"
                            text="{data.Album}"/>
                         <mx:Image id="albumImage"
                            height="45"
                            source="{data.Cover}"/>
                         <mx:TextArea
                            text="{'Renderer localVar= ' + localVar}"/>
                         <mx:TextArea
                            text="{'Application localVar= ' + outerDocument.localVar}"/>
                    </mx:VBox>
                  </fx:Component>
                </mx:itemRenderer>
            </mx:DataGridColumn>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

One use of the `outerDocument` keyword is to initialize the data provider of a control in the inline item editor. For example, you can use a web service, or other mechanism, to pass data into the application, such as the list of U.S. states. You can then initialize all ComboBox controls that are used as item editors from a single property of the application that contains the list of U.S. states.

Specifying a class name to the inline component

You can optionally specify the `className` property of the `<fx:Component>` tag to explicitly name the class generated by Flex for the inline component. By naming the class, you define a way to reference the elements in the inline component.

Creating a reusable inline item renderer or item editor

Rather than defining an inline item renderer or item editor in the definition of a component, you can define a reusable inline item renderer or item editor for use in multiple locations in your application.

For example, you use the `<fx:Component>` tag to define an inline item editor that consists of a ComboBox control for selecting the state portion of an address. You then use that inline item editor in two different DataGrid controls, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGEditorCBReUse.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="700" >

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', City: 'Boston', State: 'MA'},
                {Company: 'Allied', Contact: 'Jane Smith',
                    Phone: '617-555-3434', City: 'SanFrancisco', State: 'CA'}
            ]);
            [Bindable]
            public var initDG2:ArrayCollection = new ArrayCollection([
                {Company: 'MyCo', Contact: 'Stan Stanley',
                    Phone: '413-555-5555', City: 'Boston', State: 'MA'},
                {Company: 'YourCo', Contact: 'Dave Davis',
                    Phone: '617-555-1212', City: 'SanFrancisco', State: 'CA'}
            ]);
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:Component id="inlineEditor">
            <mx:ComboBox >
                <mx:dataProvider>
                    <fx:String>AL</fx:String>
                    <fx:String>AK</fx:String>
                    <fx:String>AR</fx:String>
                    <fx:String>CA</fx:String>
                    <fx:String>MA</fx:String>
                </mx:dataProvider>
            </mx:ComboBox>
        </fx:Component>
    </fx:Declarations>
    <mx:DataGrid id="myGrid"
        variableRowHeight="true"
        dataProvider="{initDG}"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="City"/>
            <mx:DataGridColumn dataField="State"
                width="150"
                editorDataField="selectedItem"
                itemEditor="{inlineEditor}"/>
        </mx:columns>
    </mx:DataGrid>

    <mx:DataGrid id="myGrid2"
        variableRowHeight="true"
        dataProvider="{initDG2}"
        editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="City"/>
            <mx:DataGridColumn dataField="State"
                width="150"
                editorDataField="selectedItem"
                itemEditor="{inlineEditor}"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you specify the `id` property of the inline item editor defined by the `<fx:Component>` tag. You then use data binding to specify the editor as the value of the `itemEditor` property for the two DataGrid controls.

The inline item editor or item renderer defined in the `<fx:Component>` tag appears only where you use it in the DataGrid control; otherwise, Flex ignores it when laying out your application.

Creating MX item renderers and item editor components

Defining a custom item renderer or item editor by using an MXML component gives you greater flexibility and functionality than using a drop-in item renderer or item editor. Many of the rules for defining item renderers and item editors as custom components are the same as for using inline item renderers and editors. For more information, see "Creating MX inline item renderers and editors" on page 1025.

For more information on working with custom components, see *"Custom Flex components" on page 2356*.

Creating an item renderer component

The section "Default item rendering and cell editing in MX" on page 1007 shows a DataGrid control that displays information about albums by using three text fields. You could add a visual element to your DataGrid control to make it more compelling. To do that, you modify the data provider so that it contains a URL for a JPEG image of the album cover.

The default item renderer for a DataGrid control displays data as text. To get the DataGrid control to display the image of the album cover, you use the custom item renderer defined in the RendererDGImage.mxml file, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\myComponents\RendererDGImage.mxml -->
<mx:HBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    horizontalAlign="center" >
    <mx:Image id="albumImage" height="175" source="{data.Cover}"/>
</mx:HBox>
```

The item renderer contains an Image control in an HBox container. The HBox container specifies to center the image in the container; otherwise, the image appears flush left in the cell. The Image control specifies the height of the image as 75 pixels. By default, an image has a height of 0 pixels; therefore, if you omit the height, the image does not appear.

You use data binding to associate fields of a `data` property with the controls in an item renderer or item editor. In this example, the `data` property that is passed to the item renderer contains the element of the data provider for the entire row of the DataGrid control. You then bind the Cover field of the `data` property to the Image control.

The following example illustrates using a custom item renderer with the DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\MainDGImageRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: '../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: '../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover"
                itemRenderer="myComponents.RendererDGImage"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

The DataGrid control contains a column for the album cover that uses the `itemRenderer` property to specify the name of the MXML file that contains the item renderer for that column. Now, when you run this example, the DataGrid control uses your custom item renderer for the Cover column to display an image of the album cover.

Rather than having the album name and album image in separate cells of the DataGrid control, you can use an item renderer to make them appear in a single cell, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\myComponents\RendererDGTitleImage.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    horizontalAlign="center" height="75">
    <mx:Text id="albumName"
        width="100%"
        selectable="false"
        text="{data.Album}"/>
    <mx:Image id="albumImage"
        source="{data.Cover}"/>
</mx:VBox>
```

You save this item renderer to the RendererDGTitleImage.mxml file. The DataGrid control in the main application references the item renderer, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\MainDGTitleRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: '../assets/slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: '../assets/brighten.jpg'}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist" />
            <mx:DataGridColumn dataField="Album"
                itemRenderer="myComponents.RendererDGTitleImage" />
            <mx:DataGridColumn dataField="Price"  />
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In the preceding example, you define three columns in the DataGrid control, and assign your item renderer to the second column. For an image that shows the output of this application, see "Using custom MX item renderers and item editors" on page 1009.

Creating a simple item editor component

A simple item editor component defines a single control that you use to edit a cell, and returns a single value to the list control. For an example of a simple item editor component, see "Using a component as an MX item renderer or item editor" on page 1015.

A complex item editor can contain multiple components, or can return something other than a single value to the list control. For more information, see “Advanced data display with MX item editors” on page 1048.

Overriding the data property

All components that you use in a custom item renderer or item editor that require access to the data passed to the renderer must implement the mx.core.IDataRenderer interface to define the `data` property. All Flex containers and many Flex components support this property.

Classes implement the mx.core.IDataRenderer interface by defining the `data` property as a setter and getter method, with the following signature:

```
override public function set data(value:Object):void
public function get data():Object
```

In the setter method, *value* is the `data` property passed to the item renderer.

If you define a custom component and you want to support the `data` property, your component must implement the mx.core.IDataRenderer interface. If your component is a subclass of a class that already implements the mx.core.IDataRenderer interface, you do not have to reimplement the interface.

To add programmatic logic to the controls in your item renderer or item editor that already implement the `data` property, you can override the setter or getter method for the `data` property. Typically, you override the setter method so that you can perform some operation based on the value that is passed to it.

For example, the following DataGrid control uses `true` and `false` for the values of the SalePrice field of the data provider. Although you could display these values in your DataGrid control, you can create a more compelling DataGrid control by displaying images instead, as the following item renderer shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\myComponents\RendererDGImageSelect.mxml -->
<mx:HBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    horizontalAlign="center">
    <fx:Script>
        <![CDATA[
            import mx.events.FlexEvent;
            [Embed(source="saleIcon.jpg")]
            [Bindable]
            public var sale:Class;

            [Embed(source="noSaleIcon.jpg")]
            [Bindable]
            public var noSale:Class;
            override public function set data(value:Object):void {
                if(value != null)  {
                    super.data = value;
                    if (value.SalePrice == true) onSale.source=new sale();
                    else onSale.source=new noSale();
                }
                // Dispatch the dataChange event.
                dispatchEvent(new FlexEvent(FlexEvent.DATA_CHANGE));
            }
        ]]>
    </fx:Script>
    <mx:Image id="onSale" height="20"/>
</mx:HBox>
```

In this example, you override the setter method for the HBox container. In the override, use super.data to set the data property for the base class, and then set the source property of the Image control based on the value of the SalePrice field. If the field is true, which indicates that the item is on sale, you display one icon. If the item is not on sale, you display a different icon.

The override also dispatches the dataChange event to indicate that the data property has changed. You typically dispatch this event from the setter method.

About using the creationComplete and dataChange events

Flex dispatches the creationComplete event once for a component after the component is created and initialized. Many custom components define an event listener for the creationComplete event to handle any postprocessing tasks that must be performed after the component is completely created and initialized.

However, although for an item renderer or item editor, Flex might reuse an instance of the item renderer or item editor, a reused instance of an item renderer or item editor does not redispatch the creationComplete event. Instead, you can use the dataChange event with an item renderer or item editor. Flex dispatches the dataChange event every time the data property changes. The example in the section "Accessing the listData property" on page 1019 uses the dataChange event to update the TextArea in an item renderer for a DataGrid control.

Creating an MX item renderer in ActionScript

Although you commonly create item renderers and editors in MXML, you can also create them in ActionScript, as the following example item renderer shows:

```
package myComponents {
    // myComponents/CellField.as
    import mx.controls.*;
    import mx.controls.dataGridClasses.DataGridListData;
    import mx.core.*;
    public class CellField extends TextInput
    {
        // Get the initial background color.
        public var tempBGColor:Number;

        // Define the constructor and set properties.
        public function CellField() {
            super();
            height=60;
            width=80;
            tempBGColor = getStyle("contentBackgroundColor");
            editable=false;
        }
        // Override the set method for the data property.
        override public function set data(value:Object):void {
            super.data = value;
            // Since the item renderer can be recycled,
            // set the initial background color.
            setStyle("contentBackgroundColor", tempBGColor);

            if (value != null)
            {
                text = value[DataGridListData(listData).dataField];
                if(Number(text) > 100)
                {
                    setStyle("contentBackgroundColor", 0xFF0000);
                }
                else
                {
                    setStyle("contentBackgroundColor", 0xFFFFFF);
                }
            }
            else
            {
                // If value is null, clear text.
                text= "";
            }
            super.invalidateDisplayList();
        }
    }
}
```

In the preceding example, you create a subclass of the TextInput control as your item renderer. The class must be public to be used as an item renderer or editor. This item renderer displays a red background if the value of the DataGrid cell is greater than 100.

You can use this item renderer in a DataGrid control, as the following example shows:

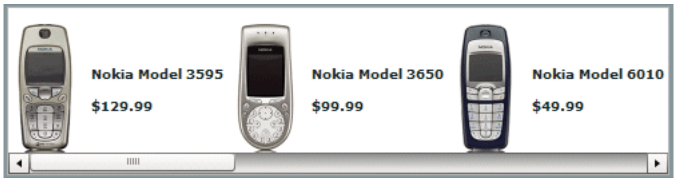
```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- itemRenderers\asRenderer\MainASItemRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="600" height="600">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Monday: 12, Tuesday: 22, Wednesday: 452, Thursday: 90},
                {Monday: 258, Tuesday: 22, Wednesday: 45, Thursday: 46},
                {Monday: 4, Tuesday: 123, Wednesday: 50, Thursday: 95},
                {Monday: 12, Tuesday: 52, Wednesday: 111, Thursday: 20},
                {Monday: 22, Tuesday: 78, Wednesday: 4, Thursday: 51}
            ]);
        ]]>
    </fx:Script>
    <mx:Text text="All cells over 100 are red" />

    <mx:DataGrid id="myDataGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Monday"
                itemRenderer="myComponents.CellField" />
            <mx:DataGridColumn dataField="Tuesday"
                itemRenderer="myComponents.CellField" />
            <mx:DataGridColumn dataField="Wednesday"
                itemRenderer="myComponents.CellField" />
            <mx:DataGridColumn dataField="Thursday"
                itemRenderer="myComponents.CellField" />
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Working with MX item renderers

Example: Using an item renderer with the MX TileList and HorizontalList controls

The TileList and HorizontalList controls display a tiled list of items. The TileList control displays items in vertical columns. The HorizontalList control displays items in horizontal rows. The TileList and HorizontalList controls are particularly useful in combination with a custom item renderer for displaying a list of images and other data.

The following image shows a HorizontalList control that is used to display a product catalog:



Each item in the HorizontalList control contains an image, a descriptive text string, and a price. The following code shows the application that displays the catalog. The item renderer for the HorizontalList control is an MXML component named Thumbnail.

```
<?xml version="1.0"?>
<!-- itemRenderers\MainTlistThumbnailRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Declarations>
        <fx:Model id="catalog" source="catalog.xml"/>
    </fx:Declarations>
    <mx:HorizontalList id="myList"
        columnWidth="125"
        rowHeight="125"
        columnCount="4"
        dataProvider="{catalog.product}"
        itemRenderer="myComponents.Thumbnail"/>
</s:Application>
```

The file catalog.xml defines the data provider for the HorizontalList control:

```
<?xml version="1.0"?>
<catalog>
    <product id="1">
        <name>Nokia Model 3595</name>
        <price>129.99</price>
        <image>../assets/products/Nokia_3595.gif</image>
        <thumbnail>../assets/products/Nokia_3595_sm.gif</thumbnail>
    </product>
    <product id="2">
        <name>Nokia Model 3650</name>
        <price>99.99</price>
        <image>../assets/products/Nokia_3650.gif</image>
        <thumbnail>../assets/products/Nokia_3650_sm.gif</thumbnail>
    </product>
    <product id="3">
        <name>Nokia Model 6010</name>
        <price>49.99</price>
        <image>../assets/products/Nokia_6010.gif</image>
        <thumbnail>../assets/products/Nokia_6010_sm.gif</thumbnail>
    </product>
    <product id="4">
        <name>Nokia Model 6360</name>
        <price>19.99</price>
        <image>../assets/products/Nokia_6360.gif</image>
        <thumbnail>../assets/products/Nokia_6360_sm.gif</thumbnail>
    </product>
    <product id="5">
        <name>Nokia Model 6680</name>
        <price>19.99</price>
        <image>../assets/products/Nokia_6680.gif</image>
        <thumbnail>../assets/products/Nokia_6680_sm.gif</thumbnail>
    </product>
    <product id="6">
        <name>Nokia Model 6820</name>
        <price>49.99</price>
        <image>../assets/products/Nokia_6820.gif</image>
        <thumbnail>../assets/products/Nokia_6820_sm.gif</thumbnail>
    </product>
</catalog>
```

The following example shows the Thumbnail.mxml MXML component. In this example, you define the item renderer to contain three controls: an Image control and two Label controls. These controls examine the `data` property that is passed to the item renderer to determine the content to display.

```
<?xml version="1.0" ?>
<!-- itemRenderers\myComponents\Thumbnail.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    horizontalAlign="center"
    verticalGap="0" borderStyle="none" backgroundColor="white" >
    <mx:Image id="image" width="60" height="60" source="{data.image}"/>
    <mx:Label text="{data.name}" width="120" textAlign="center"/>
    <mx:Label text="${data.price}" fontWeight="bold"/>
</mx:VBox>
```

For more information on the TileList and HorizontalList controls, see “MX data-driven controls” on page 943.

Example: Using an item renderer with an MX DataGrid control

The DataGrid control works with the DataGridColumn class to configure the grid. For a DataGrid control, you can specify two types of renderers: one for the cells of each column, and one for the header cell at the top of each column. To specify an item renderer for a column of a DataGrid control, you use the `DataGridColumn.itemRenderer` property. To specify an item renderer for a column header cell, you use the `DataGridColumn.headerRenderer` property.

For example, to highlight a column in the DataGrid control, you can use using an icon in the column header cell to indicate a sale price, as the following item renderer shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\RendererDGHeader.mxml -->
<mx:HBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            [Embed(source="saleIcon.jpg")]
            [Bindable]
            public var sale:Class;
        ]]>
    </fx:Script>
    <mx:Label text="Sale Price!"/>
    <mx:Image height="20" source="{sale}"/>
</mx:HBox>
```

This item renderer uses a Label control to insert the text “Sale Price!” and an Image control to insert an icon in the column header.

The following example shows the main application:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainDGHeaderRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
          import mx.collections.ArrayCollection;

          [Bindable]
          private var initDG:ArrayCollection = new ArrayCollection([
            {Artist:'Pavement', Album:'Slanted and Enchanted',
                Price:11.99, SalePrice: true },
            {Artist:'Pavement', Album:'Brighten the Corners',
                Price:11.99, SalePrice: false }
          ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
            <mx:DataGridColumn width="150" dataField="SalePrice"
                headerRenderer="myComponents.RendererDGHeader"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, the DataGrid control displays the String `true` or `false` in the column to indicate that the price is a sale price. You can also define an item renderer for that column to display a more compelling icon rather than text. For an example that uses an item renderer with a DataGrid control, see "Using custom MX item renderers and item editors" on page 1009.

Example: Using an item renderer with an MX List control

When you use a custom item renderer with a List control, you specify it using the `List.itemRenderer` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\MainListStateRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="700" width="700">
    <mx:List id="myList"
        height="180" width="250"
        variableRowHeight="true"
        itemRenderer="myComponents.RendererState">
        <mx:dataProvider>
            <fx:Object label="Alaska"
                data="Juneau"
                webPage="http://www.state.ak.us/"/>
            <fx:Object label="Alabama"
                data="Montgomery"
                webPage="http://www.alabama.gov/" />
            <fx:Object label="Arkansas"
                data="Little Rock"
                webPage="http://www.state.ar.us/"/>
        </mx:dataProvider>
    </mx:List>
</s:Application>
```

The previous example sets the `rowHeight` property to 75 pixels because the item renderer displays content that exceeds the default row height of the List control. To see an image that shows this application, see "Using custom MX item renderers and item editors" on page 1009.

The following item renderer, RendererState.mxml, displays the parts of each List item, and creates a LinkButton control which lets you open the state's web site:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\myComponents\RendererState.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            // Import Event and URLRequest classes.
            import flash.events.Event;
            import flash.net.URLRequest;

            private var u:URLRequest;

            // Event handler to open URL using
            // the navigateToURL() method.
            private function handleClick(eventObj:Event):void {
                u = new URLRequest(data.webPage);
                navigateToURL(u);
            }
        ]]>
    </fx:Script>

    <mx:HBox >
        <!-- Use Label controls to display state and capital names. -->
        <mx:Label id="State" text="State: {data.label}"/>
        <mx:Label id="Statecapital" text="Capital: {data.data}" />
    </mx:HBox>
    <!-- Define the Link control to open a URL. -->
    <mx:LinkButton id="webPage" label="Official {data.label} web page"
        click="handleClick(event);" color="blue"  />
</mx:VBox>
```

Example: Using an item renderer with an MX Tree control

For the Tree control, you use the `itemRenderer` property to specify a single renderer for all nodes of the tree. If you define a custom item renderer, you are responsible for handling the display of the entire node, including the text and icon.

One option is to define an item renderer as a subclass of the default item renderer class, the TreeItemRenderer class. You can then modify the item renderer as required by your application without implementing the entire renderer, as the following example shows:

```
package myComponents
{
    // itemRenderers/tree/myComponents/MyTreeItemRenderer.as
    import mx.controls.treeClasses.*;
    import mx.collections.*;
    public class MyTreeItemRenderer extends TreeItemRenderer
    {
        // Define the constructor.
        public function MyTreeItemRenderer() {
            super();
        }

        // Override the set method for the data property
        // to set the font color and style of each node.
        override public function set data(value:Object):void {
            super.data = value;
            if(TreeListData(super.listData).hasChildren)
            {
                setStyle("color", 0xff0000);
                setStyle("fontWeight", 'bold');
            }
            else
            {
                setStyle("color", 0x000000);
                setStyle("fontWeight", 'normal');
            }
        }

        // Override the updateDisplayList() method
        // to set the text for each tree node.
        override protected function updateDisplayList(unscaledWidth:Number,
            unscaledHeight:Number):void {

            super.updateDisplayList(unscaledWidth, unscaledHeight);
            if(super.data)
            {
                if(TreeListData(super.listData).hasChildren)
                {
                    var tmp:XMLList =
                        new XMLList(TreeListData(super.listData).item);
                    var myStr:int = tmp[0].children().length();
                    super.label.text =  TreeListData(super.listData).label +
                        "(" + myStr + ")";
                }
            }
        }
    }
}
```

For each node that has a child node, this item renderer displays the node text in red and includes the count of the number of child nodes in parentheses. The following example uses this item renderer in an application:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- itemRenderers\tree\MainTreeItemRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initCollections();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            import mx.collections.*;

            public var xmlBalanced:XMLList =
                <>
                    <node label="Containers">
                        <node label="DividedBoxClasses">
                            <node label="BoxDivider" data="BoxDivider.as"/>
                        </node>
                        <node label="GridClasses">
                            <node label="GridRow" data="GridRow.as"/>
                            <node label="GridItem" data="GridItem.as"/>
                            <node label="Other File" data="Other.as"/>
                        </node>
                    </node>
                    <node label="Data">
                        <node label="Messages">
                            <node label="DataMessage"
                                data="DataMessage.as"/>
                            <node label="SequenceMessage"
                                data="SequenceMessage.as"/>
                        </node>
                        <node label="Events">
                            <node label="ConflictEvents"
                                data="ConflictEvent.as"/>
                            <node label="CommitFaultEvent"
                                data="CommitFaultEvent.as"/>
                        </node>
                    </node>
                </>;

            [Bindable]
            public var xlcBalanced:XMLListCollection;

            private function initCollections():void {
                xlcBalanced = new XMLListCollection(xmlBalanced);
            }
        ]]>
    </fx:Script>
    <mx:Text width="400"
        text="The nodes with children are in bold red text, with the number of children in
parenthesis.)"/>
    <mx:Tree id="compBalanced"
        width="400" height="500"
        dataProvider="{xlcBalanced}"
        labelField="@label"
        itemRenderer="myComponents.MyTreeItemRenderer"/>
</s:Application>
```

Advanced data display with MX item editors

Item editors let you modify the value of a cell of a list-based control. The DataGrid, List, and Tree controls support item editors.

For an introduction to item renderers and item editors, see “MX item renderers and item editors” on page 1006.

About the MX cell editing process

The following sequence of steps occurs when a cell in a list-based control is edited:

1 User releases the mouse button while over a cell, tabs to a cell, or in another way attempts to edit a cell.

2 Flex dispatches the `itemEditBeginning` event. You can use this event to disable editing of a specific cell or cells. For more information, see “Example: Preventing a cell from being edited” on page 1062.

3 Flex dispatches the `itemEditBegin` event to open the item editor. You can use this event to modify the data passed to the item editor. For more information, see “Example: Modifying data passed to or received from an item editor” on page 1063.

4 The user edits the cell.

5 The user ends the editing session. Typically the cell editing session ends when the user removes focus from the cell.

6 Flex dispatches the `itemEditEnd` event to close the item editor and update the list-based control with the new cell data. You can use this event to modify the data returned to the cell, validate the new data, or return data in a format other than the format returned by the item editor. For more information, see “Example: Modifying data passed to or received from an item editor” on page 1063.

7 The new data value appears in the cell.

Creating an editable cell in MX

The DataGrid, List, and Tree controls include an `editable` property that you set to `true` to let users edit the contents of the control. By default, the value of the `editable` property is `false`, which means that you cannot edit the cells. For a DataGrid control, setting the `editable` property to `true` enables editing for all columns of the grid. You can disable editing for any column by setting the `DatagridColumn.editable` property to `false`.

Your list-based controls can use the default item editor (TextInput control), a custom item editor, or a custom item renderer as an editor. The `rendererIsEditor` property of the list-based controls determines whether you can use an item renderer as an item editor, as the following table shows:

rendererIsEditor property	itemRenderer property	itemEditor property
`false` (default)	Specifies the item renderer.	Specifies the item editor. Selecting the cell opens the item editor as defined by the `itemEditor` property. If the `itemEditor` property is undefined, use the default item editor (TextInput control).
`true`	Specifies the item renderer to display the cell contents. You can use the item renderer as an item editor.	Ignored.

As this table shows, the state of the `rendererIsEditor` property defines whether to use a custom item renderer as the item editor, or a custom item editor. If you set the `rendererIsEditor` property to `true`, Flex uses the item renderer as an item editor, and ignores the `itemEditor` property.

For an example, see "Example: Using an item renderer as an item editor" on page 1066.

Returning data from an MX item editor

By default, Flex expects an item editor to return a single value to the list-based control. You use the `editorDataField` property of the list-based control to specify the property of the item editor that contains the new data. Flex converts the value to the appropriate data type for the cell.

The default item editor is a TextInput control. Therefore, the default value of the `editorDataField` property is `"text"`, which corresponds to the `text` property of the TextInput control. If you specify a custom item editor, you also set the `editorDataField` property to the appropriate property of the item editor, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dropin\DropInNumStepper.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1" headerText="Order #"/>
            <mx:DataGridColumn dataField="quant"
                headerText="Qty"
                itemEditor="mx.controls.NumericStepper"
                editorDataField="value"
            />
        </mx:columns >
    </mx:DataGrid>
</s:Application>
```

In the preceding example, you use a NumericStepper control as the item editor, and therefore set the `editorDataField` property to `"value"`, the property of the NumericStepper control that contains the new cell data.

Defining a property to return data

An item editor can contain more than a single component. For example, the following item editor contains a parent VBox container with a child CheckBox control used to edit the cell:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\EditorDGCheckBox.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    backgroundColor="yellow">
    <fx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            public var cbSelected:Boolean;
        ]]>
    </fx:Script>

    <mx:CheckBox id="followUpCB" label="Follow up needed"
        height="100%" width="100%"
        selected="{data.FollowUp}"
        click="cbSelected=followUpCB.selected"
        updateComplete="cbSelected=followUpCB.selected;"/>
</mx:VBox>
```

In the preceding example, when the user selects the cell, Flex displays the CheckBox control in a yellow background, as defined by the parent VBox container. The user then selects or deselects the CheckBox control to set the new value for the cell.

To return a value to the list-based control, the VBox container defines a new property named `cbSelected`. This is necessary because you can only set the `editorDataField` property to a property of the top-level component of the item editor. That means you cannot set `editorDataField` to the `selected` property of the CheckBox control when it is a child of the VBox container.

You use the `updateComplete` event to set the value of the `cbSelected` property in case the user selects the cell to open the CheckBox control, but does not change the state of the CheckBox control.

In the following example, you use this item editor in a DataGrid control that displays a list of customer contacts. The list includes the company name, contact name, and phone number, and a column that indicates whether you must follow up with that contact.

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainDGCheckBoxEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', Date: '5/5/05' , FollowUp: true },
                {Company: 'Allied', Contact: 'Jane Smith',
                    Phone: '617-555-3434', Date: '5/6/05' , FollowUp: false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="Date"/>
            <mx:DataGridColumn dataField="FollowUp"
                width="150"
                headerText="Follow Up?"
                itemEditor="myComponents.EditorDGCheckBox"
                editorDataField="cbSelected"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Notice that the `editorDataField` property is set to `cbSelected`, the new property of the VBox container.

You also use this same mechanism with an inline item renderer that contains multiple controls. For example, you can modify the previous DataGrid example to use an inline item editor, rather than an item editor component, as the following code shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGCheckBoxEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', Date: '5/5/05' , FollowUp: true },
                {Company: 'Allied', Contact: 'Jane Smith',
                    Phone: '617-555-3434', Date: '5/6/05' , FollowUp: false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="Date"/>
            <mx:DataGridColumn dataField="FollowUp"
                width="150"
                headerText="Follow Up?"
                editorDataField="cbSelected">

                <mx:itemEditor>
                    <fx:Component>
                        <mx:VBox backgroundColor="yellow">
                            <fx:Script>
                                <![CDATA[
                                    // Define a property for returning
                                    // the new value to the cell.
                                    [Bindable]
                                    public var cbSelected:Boolean;
                                ]]>
                            </fx:Script>

                            <mx:CheckBox id="followUpCB"
                                label="Follow up needed"
                                height="100%" width="100%"
                                selected="{data.FollowUp}"
                                click="cbSelected=followUpCB.selected"/>
                        </mx:VBox>
                    </fx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Sizing and positioning an MX item editor

When an item editor appears, it appears as a pop-up control above the selected cell of the list-based control. The list-based control also hides the current value of the cell.

The CheckBox control in the previous example, “Defining a property to return data” on page 1050, sizes itself to 100% for both `width` and `height`, and the VBox container sets its `backgroundColor` style property to yellow. By sizing the item editor to the size of the cell, and by setting the background color of the container, you completely cover the underlying cell.

The following table describes the properties of the list-based controls that you can use to size the item editor:

Property	Description
`editorHeightOffset`	Specifies the height of the item editor, in pixels, relative to the size of the cell for a DataGridColumn control, or the text field of a Tree control.
`editorWidthOffset`	Specifies the width of the item editor, in pixels, relative to the size of the cell for a DataGridColum control, or the text field of a Tree control.
`editorXOffset`	Specifies the *x* location of the upper-left corner of the item editor, in pixels, relative to the upper-left corner of the cell for a DataGridColumn control, or from the upper-left corner of the text field of a Tree control.
`editorYOffset`	Specifies the *y* location of the upper-left corner of the item editor, in pixels, relative to the upper-left corner of the cell for a DataGridColumn control, or from the upper-left corner of the text field of a Tree control.

The following code modifies the definition of the DataGridColumn control from the previous section to use the `editorXOffset` and `editorYOffset` properties to move the item editor down and to the right by 15 pixels so that it has a more prominent appearance in the DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGCheckBoxEditorWithOffsets.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', Date: '5/5/05' , FollowUp: true },
                {Company: 'Allied', Contact: 'Jane Smith',
                    Phone: '617-555-3434', Date: '5/6/05' , FollowUp: false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="Date"/>
            <mx:DataGridColumn dataField="FollowUp"
```

```
                width="150"
                headerText="Follow Up?"
                editorDataField="cbSelected"
                editorXOffset="15"
                editorYOffset="15">

                <mx:itemEditor>
                    <fx:Component>
                        <mx:VBox backgroundColor="yellow">
                            <fx:Script>
                                <![CDATA[
                                    // Define a property for returning
                                    // the new value to the cell.
                                    [Bindable]
                                    public var cbSelected:Boolean;
                                ]]>
                            </fx:Script>

                            <mx:CheckBox id="followUpCB"
                                label="Follow up needed"
                                height="100%" width="100%"
                                selected="{data.FollowUp}"
                                click="cbSelected=followUpCB.selected"/>
                        </mx:VBox>
                    </fx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Creating an MX item editor that responds to the Enter key

When you use the default item editor in a list-based control, you can edit the cell value, and then press the Enter key to move focus to the next cell in the control. When you create a simple item editor that contains only a single component, and that component implements the IFocusable interface, the item editor also responds to the Enter key. The following components implement the IFocusable interface: Accordion, Button, ButtonBar, ComboBase, DateChooser, DateField, ListBase, MenuBar, NumericStepper, TabNavigator, TextArea, and TextInput.

In the following example, you define a complex item renderer with a VBox container as the top-level component and a CheckBox control as its child component:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\EditorDGCheckBox.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    backgroundColor="yellow">
    <fx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            public var cbSelected:Boolean;
        ]]>
    </fx:Script>

    <mx:CheckBox id="followUpCB" label="Follow up needed"
        height="100%" width="100%"
        selected="{data.FollowUp}"
        click="cbSelected=followUpCB.selected"
        updateComplete="cbSelected=followUpCB.selected;"/>
</mx:VBox>
```

When this item editor opens, the CheckBox control can obtain focus for editing, but pressing the Enter key does not move focus to the next cell because the parent VBox container does not implement the IFocusManagerComponent interface. You can modify this example to implement the IFocusManagerComponent interface, as the following code shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\EditorDGCheckBoxFocusable.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    backgroundColor="yellow"
    implements="mx.managers.IFocusManagerComponent" >
    <fx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            public var cbSelected:Boolean;

            // Implement the drawFocus() method for the VBox.
            override public function drawFocus(isFocused:Boolean):void {
                // This method can be empty, or you can use it
                // to make a visual change to the component.
            }
        ]]>
    </fx:Script>

    <mx:CheckBox id="followUpCB"
        label="Follow up needed"
        height="100%" width="100%"
        selected="{data.FollowUp}"
        click="cbSelected=followUpCB.selected"
        updateComplete="cbSelected=followUpCB.selected;"/>
</mx:VBox>
```

While the IFocusManagerComponent interface defines several properties and methods, the UIComponent class defines or inherits implementations for all of them except for the `drawFocus()` method. Therefore, you only have to implement that one method so that your item editor responds to the Enter key.

Using MX cell editing events

A list component dispatches the following events as part of the cell editing process: the `itemEditBeginning`, `itemEditBegin`, and the `itemEditEnd` events. The list-based controls define default event listeners for all three of these events.

You can write your own event listeners for one or more of these events to customize the editing process. When you write your own event listener, it executes before the default event listener, which is defined by the component, and then the default listener executes. For example, you can modify the data passed to or returned from an item editor. You can modify the data in the event listener for the `itemEditBegin` event. When it completes, the default event listener runs to continue the editing process.

However, you can replace the default event listener for the component with your own event listener. To prevent the default event listener from executing, you call the `preventDefault()` method from anywhere in your event listener.

Use the following events when you create an item editor:

1 `itemEditBeginning`

Dispatched when the user releases the mouse button while over a cell, tabs to a cell, or in any way attempts to edit a cell.

The list-based controls have a default listener for the `itemEditBeginning` event that sets the `editedItemPosition` property of the list-based control to the cell that has focus.

You typically write your own event listener for this event to prevent editing of a specific cell or cells. To prevent editing, call the `preventDefault()` method from within your event listener, which stops the default event listener from executing, and prevents any editing from occurring on the cell. For more information, see "Example: Preventing a cell from being edited" on page 1062.

2 `itemEditBegin`

Dispatched before an item editor opens.

The list components have a default listener for the `itemEditBegin` event that calls the `createItemEditor()` method to perform the following actions:

- Creates an item editor object, and copies the `data` property from the cell to the editor. By default, the item editor object is an instance of the TextInput control. You use the `itemEditor` property of the list-based control to specify a custom item editor class.
- Sets the `itemEditorInstance` property of the list-based control to reference the item editor object.

You can write an event listener for this event to modify the data passed to the item editor. For example, you might modify the data, its format, or other information used by the item editor. For more information, see "Example: Modifying data passed to or received from an item editor" on page 1063.

You can also create an event listener to determine which item editor you use to edit the cell. For example, you might have two different item editors. Within the event listener, you can examine the data to be edited, open the appropriate item editor by setting the `itemEditor` property to the appropriate editor, and then call the `createItemEditor()` method. In this case, first you call `preventDefault()` to stop Flex from calling the `createItemEditor()` method as part of the default event listener.

You can call the `createItemEditor()` method only from within the event listener for the `itemEditBegin` event. To create an editor at other times, set the `editedItemPosition` property to generate the `itemEditBegin` event.

3 `itemEditEnd`

Dispatched when the cell editing session ends, typically when focus is removed from the cell.

The list components have a default listener for this event that copies the data from the item editor to the data provider of the list-based control. The default event listener performs the following actions:

- Uses the `editorDataField` property of the list-based control to determine the property of the item editor that contains the new data. The default item editor is the TextInput control, so the default value of the `editorDataField` property is `"text"` to specify that the `text` property of the TextInput control contains the new cell data.
- Depending on the reason for ending the editing session, the default event listener calls the `destroyItemEditor()` method to close the item editor. For more information, see "Determining the reason for an itemEditEnd event" on page 1060.

You typically write an event listener for this event to perform the following actions:

- Modify the data returned from the item editor.

  In your event listener, you can modify the data returned by the editor to the list-based control. For example, you can reformat the data before returning it to the list-based control.

- Examine the data entered into the item editor:

  In your event listener, you can examine the data entered into the item editor. If the data is incorrect, you can call the `preventDefault()` method to stop Flex from passing the new data back to the list-based control and from closing the editor.

Cell editing event classes

Each editable list-based control has a corresponding class that defines the event object for the cell editing events, as the following table shows:

List class	Event class
DataGrid	DataGridEvent
List	ListEvent
Tree	ListEvent

Notice that the event class for the List and Tree controls is ListEvent.

When defining the event listener for a list-based control, ensure that you specify the correct type for the event object passed to the event listener, as the following example shows for a DataGrid control:

```
public function myCellEndEvent(event:DataGridEvent):void {
    // Define event listener.
}
```

Accessing cell data and the item editor in an event listener

From within an event listener, you can access the current value of the cell being edited, the new value entered by the user, or the item editor used to edit the cell.

To access the current value of a cell, you use the `editedItemRenderer` property of the list-based control. The `editedItemRenderer` property contains the data that corresponds to the cell being edited. For List and Tree controls, this property contains the data provider element for the cell. For a DataGrid control, it contains the data provider element for the entire row of the DataGrid.

To access the new cell value and the item editor, you use the `itemEditorInstance` property of the list-based control. The `itemEditorInstance` property is not initialized until after the event listener for the `cellBeginEvent` listener executes. Therefore, you typically access the `itemEditorInstance` property only from within the event listener for the `itemEditEnd` event.

The following example shows an event listener for the `itemEditEnd` event that uses these properties:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventAccessEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[

            import mx.controls.TextInput;
            import mx.events.DataGridEvent;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99 }
            ]);

            // Define event listener for the itemEditEnd event.
            private function getCellInfo(event:DataGridEvent):void {

                // Get the cell editor and cast it to TextInput.
                var myEditor:TextInput =
                    TextInput(event.currentTarget.itemEditorInstance);

                // Get the new value from the editor.
                var newVal:String = myEditor.text;

                // Get the old value.
                var oldVal:String =
            event.currentTarget.editedItemRenderer.data[event.dataField];

                // Write out the cell coordinates, new value,
                // and old value to the TextArea control.
                cellInfo.text = "cell edited.\n";
                cellInfo.text += "Row, column: " + event.rowIndex + ", " +
                    event.columnIndex + "\n";
                cellInfo.text += "New value: " + newVal + "\n";
                cellInfo.text += "Old value: " + oldVal;
            }
        ]]>
    </fx:Script>
    <mx:TextArea id="cellInfo" width="300" height="150" />

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="getCellInfo(event);" >
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, you access the item editor, and cast it to the correct editor class. The default item editor is a TextInput control, so you cast it to TextInput. If you had defined a custom item editor, you would cast it to that class. After you have a reference to the item editor, you can access its properties to obtain the new cell value.

To access the old value of the cell, you use the `editedItemRenderer` property of the DataGrid control. You then use the `dataField` property of the event object to access the `data` property for the edited column.

Determining the reason for an itemEditEnd event

A user can end a cell editing session in several ways. In the body of the event listener for the `itemEditEnd` event, you can determine the reason for the event, and then handle it accordingly.

Each event class for a list-based control defines the `reason` property, which contains a value that indicates the reason for the event. The `reason` property has the following values:

Value	Description
`CANCELLED`	Specifies that the user canceled editing and that they do not want to save the edited data. Even if you call the `preventDefault()` method from within your event listener for the `itemEditEnd` event, Flex still calls the `destroyItemEditor()` method to close the editor.
`NEW_COLUMN`	(DataGrid only) Specifies that the user moved focus to a new column in the same row. In an event listener, you can let the focus change occur, or prevent it. For example, your event listener might check that the user entered a valid value for the cell currently being edited. If not, you can prevent the user from moving to a new cell by calling the `preventDefault()` method. In this case, the item editor remains open, and the user continues to edit the current cell. If you call the `preventDefault()` method and also call the `destroyItemEditor()` method, you block the move to the new cell, but the item editor closes.
`NEW_ROW`	Specifies that the user moved focus to a new row. You handle this value for the `reason` property similar to the way you handle the `NEW_COLUMN` value.
`OTHER`	Specifies that the list-based control lost focus, was scrolled, or is somehow in a state where editing is not allowed. Even if you call the `preventDefault()` method from within your event listener for the `itemEditEnd` event, Flex still calls the `destroyItemEditor()` method to close the editor.

The following example uses the itemEditEnd event to ensure that the user did not enter an empty String in a cell. If there is an empty String, the itemEditEnd event calls preventDefault() method to prohibit the user from removing focus from the cell until the user enters a valid value. However, if the reason property for the itemEditEnd event has the value CANCELLED, the event listener does nothing:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[

            import mx.controls.TextInput;
            import mx.events.DataGridEvent;
            import mx.events.DataGridEventReason;
            import mx.formatters.NumberFormatter;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99 }
            ]);

            private var myFormatter:NumberFormatter=new NumberFormatter();

            public function formatData(event:DataGridEvent):void {
                // Check the reason for the event.
                if (event.reason == DataGridEventReason.CANCELLED)
                {
                    // Do not update cell.
                    return;
                }
                // Get the new data value from the editor.
                var newData:String=
                    TextInput(event.currentTarget.itemEditorInstance).text;
                if(newData == "")
                {
                    // Prevent the user from removing focus,
                    // and leave the cell editor open.
                    event.preventDefault();
                    // Write a message to the errorString property.
                    // This message appears when the user
                    // mouses over the editor.
                    TextInput(myGrid.itemEditorInstance).errorString=
                        "Enter a valid string.";
                }
            }

        ]]>
    </fx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="formatData(event);">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In this example, if the user's reason for the event is `CANCELLED`, the event listener does nothing.

If the reason is `NEW_COLUMN`, `NEW_ROW`, or `OTHER`, the event listener performs the following actions:

1 Checks if the new cell value is an empty String.

2 If it is an empty String, the event listener calls the `preventDefault()` method to prevent Flex from closing the editor and from updating the cell with the empty String.

3 Writes a message to the `errorString` property of the TextInput control. This message causes a red box to appear around the TextInput control, and the message appears as a tooltip when the user moves the mouse over the cell.

MX item editor examples

Example: Preventing a cell from being edited

From within an event listener for the `itemEditBeginning` event, you can inspect the cell being edited, and prevent the edit from occurring. This technique is useful when you want to prevent editing of a specific cell or cells, but allow editing of other cells.

For example, the DataGrid control uses the `editable` property to make all cells in the DataGrid control editable. You can override that for a specific column, using the `editable` property of a DataGridColumn, but you cannot enable or disable editing for specific cells.

To prevent cell editing, call the `preventDefault()` method from within your event listener for the `itemEditBeginning` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventPreventEdit.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[

            import mx.events.DataGridEvent;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99}
            ]);

            // Define event listener for the cellEdit event
            // to prohibit editing of the Album column.
            private function disableEditing(event:DataGridEvent):void {
                if(event.columnIndex==1)
                {
                    event.preventDefault();
                }
            }

        ]]>
    </fx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true"
        itemEditBeginning="disableEditing(event);" >
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Although the preceding example uses the column index, you could inspect any property of the DataGrid, or of the cell being edited, to make your decision about allowing the user to edit the cell.

Example: Modifying data passed to or received from an item editor

You can use the `itemEditBegin` and `itemEditEnd` events to examine the data passed to and from the item editor, and modify it if necessary. For example, you could reformat the data, extract a part of the data for editing, or examine the data to validate it.

In the next example, you use a NumericStepper control to edit the Price column of a DataGrid control. The `itemEditBegin` event modifies the data passed to the NumericStepper to automatically add 20% to the price when you edit it. Use the NumericStepper control to modify the updated price as necessary.

```
<?xml version="1.0"?>
<!-- itemRenderers\events\BeginEditEventAccessEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
      <![CDATA[
        import mx.events.DataGridEvent;
        import mx.controls.NumericStepper;
        import mx.collections.ArrayCollection;
        import mx.controls.listClasses.IDropInListItemRenderer;

        [Bindable]
        private var myDP:ArrayCollection = new ArrayCollection([
            {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
            {Artist:'Pavement', Album:'Crooked Rain, Crooked Rain', Price:10.99},
            {Artist:'Pavement', Album:'Wowee Zowee', Price:12.99},
            {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99},
            {Artist:'Pavement', Album:'Terror Twilight', Price:11.99}
        ]);

        // Handle the itemEditBegin event.
        private function modifyEditedData(event:DataGridEvent):void
        {
            // Get the name of the column being editted.
            var colName:String = myDataGrid.columns[event.columnIndex].dataField;

            if(colName=="Price")
            {
                // Handle the event here.
                event.preventDefault();

                // Creates an item editor.
                myDataGrid.createItemEditor(event.columnIndex,event.rowIndex);

                // All item editors must implement the IDropInListItemRenderer interface
                // and the listData property.
                // Initialize the listData property of the editor.
                IDropInListItemRenderer(myDataGrid.itemEditorInstance).listData =
                    IDropInListItemRenderer(myDataGrid.editedItemRenderer).listData;

                // Copy the cell value to the NumericStepper control.
                myDataGrid.itemEditorInstance.data = myDataGrid.editedItemRenderer.data;
                // Add 20 percent to the current price.
                NumericStepper(myDataGrid.itemEditorInstance).value +=
                    0.2 * NumericStepper(myDataGrid.itemEditorInstance).value;
           }
        }
      ]]>
    </fx:Script>
    <mx:DataGrid id="myDataGrid" dataProvider="{myDP}"
        editable="true"
        itemEditBegin="modifyEditedData(event);"
        rowHeight="60">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist" />
            <mx:DataGridColumn dataField="Album" width="130" />
            <mx:DataGridColumn dataField="Price" editorDataField="value">
                <mx:itemEditor>
                    <fx:Component>
                        <mx:NumericStepper stepSize="0.01" maximum="500"/>
                    </fx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

You could use one of the Flex formatter classes to format data returned from an item editor. In the following example, you let the user edit the Price column of a DataGrid control. You then define an event listener for the `itemEditEnd` event that uses the NumberFormatter class to format the new cell value so that it contains only two digits after the decimal point, as the following code shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >

    <fx:Script>
        <![CDATA[

            import mx.controls.TextInput;
            import mx.events.DataGridEvent;
            import mx.events.DataGridEventReason;
            import mx.formatters.NumberFormatter;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99 }
            ]);

            // Define the number formatter.
            private var myFormatter:NumberFormatter=new NumberFormatter();

            // Define the eventlistner for the itemEditEnd event.
            public function formatData(event:DataGridEvent):void {
                // Check the reason for the event.
                if (event.reason == DataGridEventReason.CANCELLED)
                {
                    // Do not update cell.
                    return;
                }
                // Get the new data value from the editor.
                var newData:String=
                    TextInput(event.currentTarget.itemEditorInstance).text;
                // Determine if the new value is an empty String.
                if(newData == "") {
                    // Prevent the user from removing focus,
                    // and leave the cell editor open.
                    event.preventDefault();
                    // Write a message to the errorString property.
                    // This message appears when the user
                    // mouses over the editor.
                    TextInput(myGrid.itemEditorInstance).errorString=
                        "Enter a valid string.";
                    return;
                }
                // For the Price column, return a value
                // with a precision of 2.
                if(event.dataField == "Price") {
                    myFormatter.precision=2;
                    TextInput(myGrid.itemEditorInstance).text=
                        myFormatter.format(newData);
                }
            }
        ]]>
    </fx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="formatData(event);" >
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Example: Using an item renderer as an item editor

If you set the `rendererIsEditor` property of the DataGrid, List, or Tree control to `true`, the control uses the default TextInput control as the item editor, or the item renderer that specifies the `itemRenderer` property. If you specify an item renderer, you must ensure that you include editable controls in it so that the user can edit values.

For example, the following item renderer displays information in the cell by using the TextInput control, and lets the user edit the cell's contents:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\MyContactEditable.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
   <fx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            [Bindable]
            public var newContact:String;
        ]]>
    </fx:Script>
    <mx:Label id="title" text="{data.label1}"/>
    <mx:Label id="contactLabel" text="Last Contacted By:"/>
    <mx:TextInput id="contactTI"
        editable="true"
        text="{data.Contact}"
        change="newContact=contactTI.text;"/>
</mx:VBox>
```

You can use this item renderer with a DataGrid control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainAppEditable.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="700" width="700">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {label1: "Order #2314", Contact: "John Doe",
                    Confirmed: false, Photo: "john_doe.jpg", Sent: false},
                {label1: "Order #2315", Contact: "Jane Doe",
                    Confirmed: true, Photo: "jane_doe.jpg", Sent: false}
            ]);
        ]]>
    </fx:Script>
    <mx:DataGrid id="myDG"
        width="500" height="250"
        dataProvider="{initDG}"
        variableRowHeight="true"
        editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Photo"
```

```
                editable="false"/>
            <mx:DataGridColumn dataField="Contact"
                width="200"
                editable="true"
                rendererIsEditor="true"
                itemRenderer="myComponents.MyContactEditable"
                editorDataField="newContact"/>
            <mx:DataGridColumn dataField="Confirmed"
                editable="true"
                rendererIsEditor="true"
                itemRenderer="mx.controls.CheckBox"
                editorDataField="selected"/>
            <mx:DataGridColumn dataField="Sent"
                editable="true"
                rendererIsEditor="false"
                itemEditor="mx.controls.CheckBox"
                editorDataField="selected"/>
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

In the previous example, you use the item renderer as the item editor by setting the `rendererIsEditor` property to `true` in the second and third columns of the DataGrid control.

Example: Using a data validator in a custom item editor

Just as you can validate data in other types of controls, you can validate data in the cells of list-based controls. To do so, you can create an item renderer or item editor that incorporates a data validator. For more information about data validators, see "Validating Data" on page 1964.

The following example shows the code for the validating item editor component. It uses a TextInput control to edit the field. In this example, you assign a PhoneNumberValidator validator to the `text` property of the TextInput control to validate the user input:

```
<?xml version="1.0"?>
<!-- itemRenderers\validator\myComponents\EditorPhoneValidator.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            [Bindable]
            public var returnPN:String;
        ]]>
    </fx:Script>

    <fx:Declarations>
        <mx:PhoneNumberValidator id="pnV"
            source="{newPN}"
            property="text"
            trigger="{newPN}"
            triggerEvent="change"
            required="true"/>
    </fx:Declarations>

    <mx:TextInput id="newPN"
        text="{data.phone}"
        updateComplete="returnPN=newPN.text;"
        change="returnPN=newPN.text;"/>
</mx:VBox>
```

If the user enters an incorrect phone number, the PhoneNumberValidator draws a red box around the editor and shows a validation error message when the user moves the mouse over it.

The following example shows the code for the application that uses this item editor:

```
<?xml version="1.0" ?>
<!-- itemRenderers\validator\MainDGValidatorEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {name: 'Bob Jones', phone: '413-555-1212',
                    email: 'bjones@acme.com'},
                {name: 'Sally Smith', phone: '617-555-5833',
                    email: 'ssmith@acme.com'},
            ]);
        ]]>
    </fx:Script>

    <mx:DataGrid id="dg"
        width="500" height="200"
        editable="true"
        dataProvider="{initDG}">
        <mx:columns>
            <mx:DataGridColumn dataField="name"
                headerText="Name" />
            <mx:DataGridColumn dataField="phone"
                headerText="Phone"
                itemEditor="myComponents.EditorPhoneValidator"
                editorDataField="returnPN"/>
            <mx:DataGridColumn dataField="email" headerText="Email" />
        </mx:columns>
    </mx:DataGrid>
</s:Application>
```

Examples using MX item editors with the list-based controls

Example: Using an item editor with an MX DataGrid control

For examples of using item editors with the DataGrid control, see “Returning data from an MX item editor” on page 1049.

Example: Using a custom item editor with an MX List control

You can add an item editor to a List control to let users edit the information for each state, as the following item editor shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\myComponents\EditorStateInfo.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <mx:TextInput id="newLabel" text="{data.label}" />
    <mx:TextInput id="newData" text="{data.data}" />
    <mx:TextInput id="newWebPage" text="{data.webPage}" />
</mx:VBox>
```

You define an item editor that contains three TextInput controls that let the user edit the state name, capital, or web address. This item editor returns three values, so you write an event listener for the itemEditEnd event to write the values to the data provider of the List control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\MainListStateRendererEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="700" width="700">
    <fx:Script>
        <![CDATA[
            import mx.events.ListEvent;
            import myComponents.EditorStateInfo;

            // Define the event listener.
            public function processData(event:ListEvent):void {
                // Disable copying data back to the control.
                event.preventDefault();
                // Get new label from editor.
                myList.editedItemRenderer.data.label =
    EditorStateInfo(event.currentTarget.itemEditorInstance).newLabel.text;
                // Get new data from editor.
                myList.editedItemRenderer.data.data =
    EditorStateInfo(event.currentTarget.itemEditorInstance).newData.text;
                // Get new webPage from editor.
                myList.editedItemRenderer.data.webPage =
    EditorStateInfo(event.currentTarget.itemEditorInstance).newWebPage.text;
                // Close the cell editor.
                myList.destroyItemEditor();

                // Notify the list control to update its display.
        myList.dataProvider.itemUpdated(myList.editedItemRenderer);
            }
        ]]>
    </fx:Script>
    <mx:List id="myList"
        height="180" width="250"
        editable="true"
        itemRenderer="myComponents.RendererState"
        itemEditor="myComponents.EditorStateInfo"
        variableRowHeight="true"
        itemEditEnd="processData(event);">
        <mx:dataProvider>
            <fx:Object label="Alaska"
                data="Juneau"
                webPage="http://www.state.ak.us/"/>
            <fx:Object label="Alabama"
                data="Montgomery"
                webPage="http://www.alabama.gov/" />
            <fx:Object label="Arkansas"
                data="Little Rock"
                webPage="http://www.state.ar.us/"/>
        </mx:dataProvider>
    </mx:List>
</s:Application>
```

This example uses the RendererState.mxml renderer. You can see that renderer in the section “Example: Using an item renderer with an MX List control” on page 1043.

Using a DateField or ComboBox control as a drop-in item editor

You can use a DateField or ComboBox control as a drop-in item editor with the List control. However, when the data provider is a collection of Objects, you have to set the `labelField` property of the List control to the name of the field in the data provider modified by the DateField or ComboBox control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\ListEditorDateField.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[

            import mx.collections.*;
            import mx.controls.DateField;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var catalog:ArrayCollection = new ArrayCollection([
                {confirmed: new Date(), Location: "Spain"},
                {confirmed: new Date(2006,0,15), Location: "Italy"},
                {confirmed: new Date(2004,9,24), Location: "Bora Bora"},
                {confirmed: new Date(), Location: "Vietnam"}
            ]);
        ]]>
    </fx:Script>
    <mx:List id="myList"
        width="300" height="300"
        rowHeight="50"
        dataProvider="{catalog}"
        editable="true"
        labelField="confirmed"
        itemEditor="mx.controls.DateField"
        editorDataField="selectedDate"/>
</s:Application>
```

In this example, you specify `"confirmed"` as the value of the `labelField` property to specify that the DateField control modifies that field of the data provider.

Example: Using a custom item editor with an MX Tree control

In a Tree control, you often display a single label for each node in the tree. However, the data provider for each node may contain additional data that is normally hidden from view.

In this example, you use the Tree control to display contact information for different companies. The Tree control displays the company name as a branch node, and different department names within the company as leaf nodes. Selecting any node opens a custom item editor that lets you modify the phone number or contact status of the company or for any department in the company.

The top node in the Tree control is not editable. Therefore, this example uses the `itemEditBeginning` event to determine if the user selects the top node. If selected, the event listener for the `itemEditBeginning` event prevents editing from occurring, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\tree\MainTreeEditor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="600" height="600">
    <fx:Script>
        <![CDATA[

        import mx.events.ListEvent;
        import myComponents.TreeEditor;

        private var contacts1:Object =
          {label: "top", children: [
            {label: "Acme", status: true, phone: "243-333-5555", children: [
              {label: "Sales", status: true, phone: "561-256-5555"},
              {label: "Support", status: false, phone: "871-256-5555"}
            ]},
            {label: "Ace", status: true, phone: "444-333-5555", children: [
              {label: "Sales", status: true, phone: "232-898-5555"},
              {label: "Support", status: false, phone: "977-296-5555"},
            ]},
            {label: "Platinum", status: false, phone: "521-256-5555"}
        ]};

        private function initCatalog(cat:Object):void {
            myTree.dataProvider = cat;
        }

        // Define the event listener for the itemEditBeginning event
        // to disable editing when the user selects
        // the top node in the tree.
        private function disableEditing(event:ListEvent):void {
            if(event.rowIndex==0) {
                event.preventDefault();
            }
        }

        // Define the event listener for the itemEditEnd event
        // to copy the updated data back to the data provider
        // of the Tree control.
        public function processData(event:ListEvent):void {

            // Disable copying data back to the control.
            event.preventDefault();
            // Get new phone number from editor.
            myTree.editedItemRenderer.data.phone =
    TreeEditor(event.currentTarget.itemEditorInstance).contactPhone.text;
            // Get new status from editor.
            myTree.editedItemRenderer.data.status =
    TreeEditor(event.currentTarget.itemEditorInstance).confirmed.selected;
            // Close the cell editor.
            myTree.destroyItemEditor();

            // Notify the list control to update its display.
    myTree.dataProvider.itemUpdated(myTree.editedItemRenderer);
        }
        ]]>
    </fx:Script>

    <mx:Tree id="myTree"
        width="400" height="400"
        editable="true"
        itemEditor="myComponents.TreeEditor"
        editorHeightOffset="75" editorWidthOffset="-100"
        editorXOffset="40" editorYOffset="30"
        creationComplete="initCatalog(contacts1);"
        itemEditBeginning="disableEditing(event);"
        itemEditEnd="processData(event);"/>
</s:Application>
```

You specify the custom item editor using the `itemEditor` property of a Tree control. You also use the `editorHeightOffset`, `editorWidthOffset`, `editorXOffset`, and `editorYOffset` properties to position the item editor.

The following item editor, defined in the file TreeEditor.mxml, lets you edit the data associated with each item in a Tree control:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- itemRenderers/tree/myComponents/TreeEditor.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            // Define variables for the new data.
            public var newPhone:String;
            public var newConfirmed:Boolean;
        ]]>
    </fx:Script>
    <!-- Display item label.-->
    <mx:Label text="{data.label}"/>
    <!-- Display the text 'Phone:' and let the user edit it.-->
    <mx:HBox>
        <mx:Text text="Phone:"/>
        <mx:TextInput id="contactPhone"
            width="150"
            text="{data.phone}"
            change="newPhone=contactPhone.text;"/>
    </mx:HBox>
    <!-- Display the status using a CheckBox control
        and let the user edit it.-->
    <mx:CheckBox id="confirmed"
        label="Confirmed"
        selected="{data.status}"
        click="newConfirmed=confirmed.selected;"/>
</mx:VBox>
```

Introduction to charts

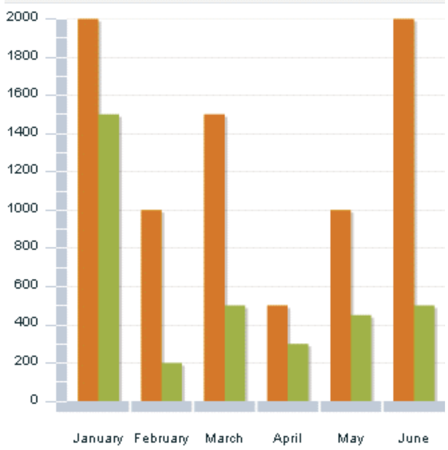
Displaying data in a chart or graph can make data interpretation much easier for users of the applications that you develop with the Adobe® Flex® product line. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other type of chart using colors, captions, and a two-dimensional representation of your data.

About charting

Data visualization lets you present data in a way that simplifies data interpretation and data relationships. Charting is one type of data visualization in which you create two-dimensional representations of your data. Flex supports the most common types of two-dimensional charts (such as bar, column, and pie charts) and gives you a great deal of control over the appearance of charts.

A simple chart shows a single data series, where a series is a group of related data points. For example, a data series might be monthly sales revenues or daily occupancy rates for a hotel.

Another chart might add a second data series. For example, you might include the percentage growth of profits over the same four business quarters. The following chart shows two data series—one for sales and one for profit.



Using the charting controls

Flex charting controls lets you create some of the most common chart types, and also lets you customize the appearance of your charts. The charting controls are located in the mx.charts.* package.

The following table lists the supported chart types, the name of the control class, and the name of the series class that you use to define what data appears in each chart.

Chart type	**Chart control class**	**Chart series class**
Area	AreaChart	AreaSeries
Bar	BarChart	BarSeries
Bubble	BubbleChart	BubbleSeries
Candlestick	CandlestickChart	CandlestickSeries
Column	ColumnChart	ColumnSeries
HighLowOpenClose	HLOCChart	HLOCSeries

Chart type	Chart control class	Chart series class
Line	LineChart	LineSeries
Pie	PieChart	PieSeries
Plot	PlotChart	PlotSeries

All chart controls, except the PieChart class, are subclasses of the CartesianChart class. Cartesian charts are charts that typically represent a set of data points in rectangular-shaped, two-dimensional space. The PieChart class is a subclass of the PolarChart class, which represents data in circular space.

All chart controls inherit basic charting characteristics from the ChartBase class.

A chart control typically has the following structure in MXML:

```
<mx:chart_type>
    <!-- Define one or more series. -->
    <mx:series>
        <mx:Series1/>
        ...
    </mx:series>

    <!-- Define the axes. -->
    <mx:horizontalAxis>
        <mx:axis_type/>
    </mx:horizontalAxis>
    <mx:verticalAxis>
        <mx:axis_type/>
    </mx:verticalAxis>

    <!-- Style the axes and ticks marks. -->
    <mx:horizontalAxisRenderers>
        <mx:AxisRenderer/>
    </mx:horizontalAxisRenderers>
    <mx:verticalAxisRenderers>
        <mx:AxisRenderer/>
    </mx:verticalAxisRenderers/>

    <!-- Add grid lines and other elements to the chart. -->
    <mx:annotationElements>
        <array of elements/>
    </mx:annotationElements>
    <mx:backgroundElements>
        <array of elements/>
    </mx:backgroundElements/>
</mx:chart_type>

<!-- Optionally define the legend. -->
<mx:Legend/>
```

The following table describes the parts of the chart in more detail:

Part	**Description**
Chart	(Required) Defines a data provider for the chart. Also defines the chart type and sets data tips, mouse sensitivity, gutter styles, and axis styles. This is the top-level tag for a chart control. All other tags are child tags of this tag.
Series	(Required) Defines one or more data series to be displayed on the chart. Also sets the strokes, fills, and renderers (or *skins*) of the data series, as well as the strokes and fills used by the chart's legend for each series. You can use different series types in the same chart. Each series in a chart can have its own data provider, or they can share the data provider set on the chart object.
Axes	Sets the axis type (numeric or category) for each axis. Also defines the axis labels, titles, and style properties such as padding.
Axes renderers	(Optional) Sets tick placement and styles, enables or disables labels, and defines axis lines, label rotation, and label gap.
Elements	(Optional) Defines grid lines and extra elements to appear on the chart.

For each chart type, Flex supplies a corresponding chart control and chart series. The chart control defines the chart type, the data provider that supplies the chart data, the grid lines, the text for the chart axes, and other properties specific to the chart type. The `dataProvider` property of the chart control determines what data the chart uses.

A *data provider* is a collection of objects. It can be an Array of objects or any object that implements the collections API. A data provider can also be an XMLList object with XML nodes, such as the result of an E4X query.

The chart components use a flat, or list-based, data provider similar to a one-dimensional array. The data provider can contain objects such as Strings and Numbers, or even other objects. For more information on supplying chart data, see “Defining chart data” on page 1090.

You use the chart series to identify which data from the data provider the chart displays. A data provider can contain more data than you want to show in your chart, so you use the chart's series to specify which points you want to use from the data provider. You can specify more than one series. You can also use the chart series to define the appearance of the data in the chart.

All chart series inherit the data provider from the chart unless they have a data provider explicitly set on themselves. If you set the value of the `dataProvider` property on the chart control, you are not required to set the property value on the series. You can, however, define different data providers for each series in a chart.

For example, to create a pie chart, you use the PieChart control with the PieSeries chart series. To create an area chart, you use the AreaChart control with the AreaSeries chart series, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicAreaOneSeries.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
            {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
            {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
        ]);
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Area Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"/>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

This example defines an array containing a single `<mx:AreaSeries>` tag. The `<mx:AreaSeries>` tag specifies the single data series that is displayed in the chart.

You can add a second `<mx:AreaSeries>` tag to display two data series, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicArea.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Area Chart">
      <s:layout>
          <s:VerticalLayout/>
      </s:layout>
     <mx:AreaChart id="myChart"
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
               categoryField="month"
               displayName="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries yField="profit" displayName="Profit"/>
           <mx:AreaSeries yField="expenses" displayName="Expenses"/>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You are not required to define a data provider on the chart control itself. Instead, you can define a different data provider for each element in a chart control that requires one. Each series can have its own data provider, as well as each axis, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/MultipleDataProviders.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var profit04:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000},
        {Month:"Feb", Profit:1000},
        {Month:"Mar", Profit:1500}
     ]);
     [Bindable]
     public var profit05:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2200},
        {Month:"Feb", Profit:1200},
        {Month:"Mar", Profit:1700}
     ]);
     [Bindable]
     public var profit06:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2400},
        {Month:"Feb", Profit:1400},
        {Month:"Mar", Profit:1900}
     ]);
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis dataProvider="{profit04}" categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
```

```
           <mx:ColumnSeries
                dataProvider="{profit04}"
                yField="Profit"
                xField="Month"
                displayName="2004"/>
           <mx:ColumnSeries
                dataProvider="{profit05}"
                yField="Profit"
                xField="Month"
                displayName="2005"/>
           <mx:ColumnSeries
                dataProvider="{profit06}"
                yField="Profit"
                xField="Month"
                displayName="2006"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To dynamically size the chart to the size of its container, set the `width` and `height` attributes to a percentage value, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicBarSize.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Script>
    <![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
    ]]>
  </fx:Script>
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        height="100%"
        width="100%"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"/>
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
</s:Application>
```

If the chart is inside a container (such as a Panel), set the size of the chart's parent containers to percentage values too.

About the series classes

The chart series classes let you specify what data to render in a chart control. All series classes are subclasses of the Series class.

Each chart type has its own series class; for example, a BarChart control has a BarSeries class that defines the data to render in the BarChart. A PieChart control has a PieSeries.

The primary purpose of a series is to define what data to render in the chart. You use the series to define what field in a data provider the chart should use to render chart items on the X and Y axes. You use the `xField` property (for the horizontal axis) and the `yField` property (for the vertical axis) to define these fields.

Each series is made up of an Array of series items. The classes that define the series items are specific to each series type. For example, a BarSeries is made up of BarSeriesItem objects. A ColumnSeries is made up of ColumnSeriesItem objects. The series items encapsulate all the information about the particular data point, including the minimum value, the x value, and the y value.

When you create a new series, you typically define the `displayName` of that series. This property represents the series to the user in labels such as DataTip objects.

A BarChart typically specifies one or more BarSeries objects that define what set of bars to render in the chart. By default, bars and columns are clustered. However, you can also define alternate ways to group, or "stack", series in the chart control. For example, AreaSeries, ColumnSeries, and BarSeries can be stacked or overlaid. They can also render as 100% charts. You can further control how multiple series are grouped by using sets. For example, for a group of BarSeries objects, you use the BarSet class; for a group of ColumnSeries objects, you use the ColumnSet class. For more information on grouping series, see "Stacking charts" on page 1308.

Most charts use only one kind of series. However, you can specify multiple types of series for a chart. For example, you can create a chart that has a series of bars in addition to a trend line that "floats" over it. This is useful for rendering trend-lines or showing different types of data on a single chart for comparison analysis. For more information, see "Using multiple data series" on page 1163.

You use the series classes to define the appearance of the chart items. You can change the fill of all series items by using the `fill` property on the series. In addition, you can define the fill of each item in a series by using the `fills` property. You can also customize the fill that each chart item has based on its value by using the `fillFunction` on the series. For more information “Using fills with chart controls” on page 1197.

You can also apply filters to series to give them effects such as drop shadows, blurs, and glows. For more information, see “Using filters with chart controls” on page 1213.

You can add data labels to series items. You do this by setting the value of the `labelPosition` property on the series. For most series, possible values of `labelPosition` are `inside` and `outside`, which draw the labels inside the chart item and outside the chart item, respectively. For a PieSeries, you can also set the `labelPosition` property to other values that include `callout` and `insideWithCallout`.

You can customize data labels by using a `labelFunction`. This callback function takes arguments that define the series item and returns a String that is then rendered on the series item.

For information about adding data labels to your charts, see “Using data labels” on page 1277.

Series also let you set a `minField` value. This property lets you specify a minimum value that the series displays. For more information, see “Using the minField property” on page 1306.

About the axis classes

Flex charting controls support the following types of axes:

CategoryAxis CategoryAxis class maps a set of values (such as stock ticker symbols, state names, or demographic categories) to the axis. You use the `<mx:CategoryAxis>` tag to define axis labels that are grouped by logical associations and that are not necessarily numeric. For example, the month names used in the chart in “About charting” on page 1075 could be defined as a CategoryAxis class.

LinearAxis A LinearAxis class maps numeric data to the axis. You use the `<mx:LinearAxis>` child tag of the `<mx:horizontalAxis>` or `<mx:verticalAxis>` tags to customize the range of values displayed along the axis, and to set the increment between the axis labels of the tick marks.

LogAxis A LogAxis class maps numeric data to the axis logarithmically. You use the `<mx:LogAxis>` child tag of the `<mx:horizontalAxis>` or `<mx:verticalAxis>` tags. Labels on the logarithmic axis are even powers of 10.

DateTimeAxis A DateTimeAxis class maps time-based values, such as hours, days, weeks, or years, along a chart axis. You use the `<mx:DateTimeAxis>` tag to define the axis labels.

The DateTimeAxis, LogAxis, and LinearAxis are all of type NumericAxis, because they are used to represent numeric values. In many cases, you are required to define only one axis as being a NumericAxis or a CategoryAxis. Flex assumes that all axes not explicitly defined are of type LinearAxis. However, to use decorations such as DataTip labels and legends, you might be required to explicitly define both axes.

There are exceptions. For a PlotChart control, both axes are considered a LinearAxis, because the data point is the intersection of two coordinates. So, you are not required to specify either axis, although you can do so to provide additional settings, such as minimum and maximum values. When you create PieChart controls, you also do not specify either axis, because PieChart controls use a single set of data points to draw wedges that represent a percentage of the whole. In PieChart controls, you define a `nameField` on the chart’s data series, rather than a `categoryField` or `name` on the axes for labels and legends.

Each axis can have one or more corresponding AxisRenderer objects (specified by the `horizontalAxisRenderers` or `verticalAxisRenderers` properties) that define the appearance of axis labels and tick marks. In addition to defining formats, you can use an AxisRenderer class to customize the value of the axis labels. For more information, see “Formatting charts” on page 1169.

The appearance and contents of *axis labels* are defined by the `<mx:horizontalAxis>` (x-axis) and `<mx:verticalAxis>` (y-axis) tags and the renderers for these tags (`<mx:AxisRenderer>` tags within the `<mx:horizontalAxisRenderers>` and `<mx:verticalAxisRenderers>` tags). These tags not only define the data ranges that appear in the chart, but also map the data points to their names and labels. This mapping has a large impact on how the Data Management Service chart renders the values of DataTip labels, axis labels, and tick marks.

By default, Flex uses the chart type and orientation to calculate the labels that appear along the x-axis and y-axis of the chart. The labels of a column chart, for example, have the following default values:

The x-axis The minimum number of labels is 0, and the maximum is the number of items in the data series that is being charted.

The y-axis The minimum value on the y-axis is small enough for the chart data, and the maximum value is large enough based to accomodate the chart data.

For more information about chart axes, see "Working with axes" on page 1247.

About charting events

The chart controls include events that accommodate user interaction with data points in charts. These events are described in "Events and effects in charts" on page 1324.

Creating charts in ActionScript

You can create, destroy, and manipulate charts using ActionScript just as you can any other Flex component.

When working in Script blocks or in separate ActionScript class files, you must be sure to import all appropriate classes. The following set of import statements defines the most common cases:

```
import mx.collections.*;
import mx.charts.*;
import mx.charts.series.*;
import mx.charts.renderers.*;
import mx.charts.events.*;
```

To create a chart in ActionScript, use the `new` keyword. You can set properties on the chart object as you would in MXML. You assign a data provider with the `dataProvider` property. To add a data series to the chart, you define a new data series of the appropriate type. To apply the series to your chart, use the chart's `series` property. You can specify the category axis settings using the CategoryAxis class. The following example defines a BarChart control with two series:

```
<?xml version="1.0"?>
<!-- charts/CreateChartInActionScript.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init()"
    height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.BarChart;
     import mx.charts.series.BarSeries;
     import mx.charts.CategoryAxis;
     import mx.charts.Legend;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
     public var myChart:BarChart;
     public var series1:BarSeries;
     public var series2:BarSeries;
     public var legend1:Legend;
     public function init():void {
        /* Create the chart object and set some
           basic properties. */
        myChart = new BarChart();
        myChart.showDataTips = true;
        myChart.dataProvider = expenses;
        /* Define the category axis. */
        var vAxis:CategoryAxis = new CategoryAxis();
        vAxis.categoryField = "Month" ;
        vAxis.dataProvider =  expenses;
        myChart.verticalAxis = vAxis;
        /* Add the series. */
        var mySeries:Array=new Array();
        series1 = new BarSeries();
        series1.xField="Profit";
        series1.yField="Month";
        series1.displayName = "Profit";
        mySeries.push(series1);
        series2 = new BarSeries();
        series2.xField="Expenses";
        series2.yField="Month";
        series2.displayName = "Expenses";
        mySeries.push(series2);
        myChart.series = mySeries;
        /* Create a legend. */
        legend1 = new Legend();
        legend1.dataProvider = myChart;
        /* Attach chart and legend to the display list. */
        p1.addElement(myChart);
        p1.addElement(legend1);
     }
  ]]></fx:Script>
  <s:Panel id="p1" title="BarChart Created in ActionScript">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  </s:Panel>
</s:Application>
```

This example replaces the existing Array of series with the new series.

You can use a similar technique to add data series to your charts rather than replacing the existing ones. The following example creates two ColumnSeries and sets their data providers. It then creates an Array that holds the existing chart series, and pushes the new series into that Array. Finally, it sets the value of the chart's `series` property to be the new Array of series.

```
<?xml version="1.0"?>
<!-- charts/AddingSeries.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_fred.send();srv_fe.send();srv_strk.send();"
    height="600">
    <fx:Declarations>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
         <!-- View source of the following pages to see the structure of the data that Flex
uses in this example. -->
       <mx:HTTPService id="srv_fred" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FRED"/>
         <mx:HTTPService id="srv_fe" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FE"/>
       <mx:HTTPService id="srv_strk" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=STRK"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import mx.charts.series.ColumnSeries;
        import mx.collections.ArrayCollection;
        private var series1:ColumnSeries;
        private var series2:ColumnSeries;
        private function addMoreSeries():void {
            if (!series1 || !series2) {
                series1 = new ColumnSeries();
                series1.dataProvider = srv_fe.lastResult.data.result;
                series1.yField = "close";
                series1.xField = "date";
                series1.displayName = "FE";
                series2 = new ColumnSeries();
                series2.dataProvider = srv_strk.lastResult.data.result;
                series2.yField = "close";
                series2.xField = "date";
                series2.displayName = "STRK";
                var currentSeries:Array = myChart.series;
                currentSeries.push(series1);
                currentSeries.push(series2);
                myChart.series = currentSeries;
            }
        }
        private function resetApp():void {
            myChart.series = [ series0 ];
            series1 = null;
            series2 = null;
        }
    ]]></fx:Script>
    <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart" showDataTips="true">
            <mx:horizontalAxis>
                <mx:DateTimeAxis dataUnits="days"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries dataProvider="{srv_fred.lastResult.data.result}"
                    id="series0"
                    yField="close"
                    xField="date"
                    displayName="FRED"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
    <s:HGroup>
        <s:Button id="b1" label="Add More Series To Chart" click="addMoreSeries()"/>
        <s:Button id="b2" label="Reset" click="resetApp()"/>
    </s:HGroup>
</s:Application>
```

By using ActionScript, you can also define a variable number of series for your charts. The following example uses E4X syntax to extract an Array of unique names from the data. It then iterates over this Array and builds a new LineSeries for each name.

```
<?xml version="1.0"?>
<!-- charts/VariableSeries.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();"
    height="600">

  <fx:Script><![CDATA[
        import mx.charts.series.LineSeries;
        import mx.charts.DateTimeAxis;
        [Bindable]
        private var myXML:XML =
            <dataset>
            <item>
                <who>Tom</who>
                <when>08/22/2006</when>
                <hours>5.5</hours>
            </item>
            <item>
                <who>Tom</who>
                <when>08/23/2006</when>
                <hours>6</hours>
            </item>
            <item>
                <who>Tom</who>
                <when>08/24/2006</when>
                <hours>4.75</hours>
            </item>
            <item>
                <who>Dick</who>
                <when>08/22/2006</when>
                <hours>6</hours>
            </item>
            <item>
                <who>Dick</who>
                <when>08/23/2006</when>
                <hours>8</hours>
            </item>
            <item>
                <who>Dick</who>
                <when>08/24/2006</when>
                <hours>7.25</hours>
            </item>
            <item>
                <who>Jane</who>
                <when>08/22/2006</when>
                <hours>6.5</hours>
            </item>
            <item>
                <who>Jane</who>
                <when>08/23/2006</when>
                <hours>9</hours>
            </item>
            <item>
                <who>Jane</who>
                <when>08/24/2006</when>
                <hours>3.75</hours>
            </item>
            </dataset>;
        public function initApp():void {
            var wholist:Array = new Array();
            for each(var property:XML in myXML.item.who) {
                // Create an Array of unique names.
                if (wholist[property] != property)
                    wholist[property] = property;
            }
            // Iterate over names and create a new series
            // for each one.
            for (var s:String in wholist) {
                // Use all items whose name matches s.
                var localXML:XMLList = myXML.item.(who==s);
                // Create the new series and set its properties.
                var localSeries:LineSeries = new LineSeries();
                localSeries.dataProvider = localXML;
                localSeries.yField = "hours";
                localSeries.xField = "when";
                // Set values that show up in dataTips and Legend.
                localSeries.displayName = s;
                // Back up the current series on the chart.
                var currentSeries:Array = myChart.series;
                // Add the new series to the current Array of series.
                currentSeries.push(localSeries);
                // Add the new Array of series to the chart.
                myChart.series = currentSeries;
            }
            // Create a DateTimeAxis horizontal axis.
            var hAxis:DateTimeAxis = new DateTimeAxis();
            hAxis.dataUnits = "days";
            // Set this to false to display the leftmost label.
            hAxis.alignLabelsToUnits = false;
            // Take the date in its current format and create a Date
            // object from it.
            hAxis.parseFunction = createDate;
            myChart.horizontalAxis = hAxis;
        }
        public function createDate(s:String):Date {
            // Reformat the date input to create Date objects
            // for the axis.
```

```
            var a:Array = s.split("/");

            // The existing String s is in the format "MM/DD/YYYY".
            // To create a Date object, you pass "YYYY,MM,DD",
            // where MM is zero-based, to the Date() constructor.
            var newDate:Date = new Date(a[2],a[0]-1,a[1]);
            return newDate;
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart with Variable Number of Series">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:LineChart id="myChart" showDataTips="true"/>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Defining chart data

The chart controls have a `dataProvider` property that defines the data for the chart.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple charts from the same data provider, switch data providers for a chart at run time, and modify the data provider so that changes are reflected by all charts using the data provider.

You can set the data provider on the chart, in which case all series inherit that data provider. Or you can set the data provider on each series in the chart.

Using chart data

To use the data from a data provider in your chart control, you map the `xField` and `yField` properties of the chart series to the fields in the data provider. The `xField` property defines the data for the horizontal axis, and the `yField` property defines the data for the vertical axis.

In ActionScript object syntax, assume your data provider has the following structure:

```
{month: "Feb", profit: 1000, expenses: 200, amount: 60}
```

In XML, the data might appear like the following:

```
<data>
    <result month="Feb">
        <profit>1000</profit>
        <expenses>200</expenses>
        <amount>60</amount>
    </result>
</data>
```

You can use the `profit` and `expenses` fields and ignore the `month` field by mapping the `xField` property of the series object to one field and the `yField` property of the series object to another field, as the following example shows:

```
<mx:PlotSeries xField="profit" yField="expenses"/>
```

The result is that each data point is the intersection of the `profit` and `expenses` fields from the data provider.

To place the data points into a meaningful grouping, you can choose a separate property of the data provider as the `categoryField`. In this case, to sort each data point by month, you map the `month` field to the `categoryField` property of the horizontal axis:

```
<mx:horizontalAxis>
    <mx:CategoryAxis
        dataProvider="{srv.lastResult.data.result}"
        categoryField="month"
    />
</mx:horizontalAxis>
```

In some cases, depending on the type of chart and the type of data you are representing, you use either the `xField` property or the `yField` property to define the data series. In a ColumnChart control, for example, the `yField` property defines the height of the column. You do not have to specify an `xField` property. To get an axis label for each column, you specify a `categoryField` property for the `horizontalAxis`.

The data provider can contain complex objects, or objects within objects. For example, a data provider object can have the following structure:

```
{month: "Aug", close: {High:45.87,Low:12.2}, open:25.19}
```

In this case, you cannot simply refer to the field of the data provider by using a `categoryField`, `xField`, or similar flat naming convention. Rather, you use the `dataFunction` of the series or axis to drill down into the data provider. For more information on working with complex data, see "Structure of chart data" on page 1113.

When you use chart data, keep the following in mind:

- You usually match a series with a data provider field if you want to display that series. However, this is not always true. If you do not specify an `xField` for a ColumnSeries, Flex assumes the index is the value. If you do not specify a `yField`, Flex assumes the data provider is a collection of *y* values, rather than a collection of objects that have *y* values. For example, the following series renders correctly for a ColumnChart control:

  ```
  <mx:ColumnSeries dataProvider="{[1,2,3,4,5]}"/>
  ```

- Some series use only one field from the data provider, while others can use two or more. For example, you specify only a `field` property for a PieSeries object, but you can specify an `xField` and a `yField` for a PlotSeries object and an `xField`, `yField`, and `radiusField` for a BubbleSeries object.
- Most of the series can determine suitable defaults for their nonprimary dimensions if no field is specified. For example, if you do not explicitly set an `xField` for the ColumnSeries, LineSeries, and AreaSeries, Flex maps the data to the chart's categories in the order in which the data appears in the data provider. Similarly, a BarSeries maps the data to the categories if you do not set a `yField`.

For a complete list of the fields that each data series can use, see the data series's class entry in *ActionScript 3.0 Reference for the Adobe Flash Platform*. For more information on data providers, see "Data provider controls" on page 645.

Sources of chart data

You can supply data to a data provider in the following ways:

- Define it in a `<fx:Script>` block.
- Define it in an external XML, ActionScript, or text file.
- Return it by using a WebService call.
- Return it by using a RemoteObject component.
- Return it by using an HTTPService component.
- Define it in MXML.

There are some limitations on the structure of the chart data, and how to reference chart data if it is constructed with complex objects. For more information, see “Structure of chart data” on page 1113.

For more information on data providers, see “Data providers and collections” on page 898.

Using static Arrays as data providers

Using a static Array of objects for the data provider is the simplest approach. You typically create an Array of objects, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfObjectsDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     [Bindable]
     private var expenses:Array = [
        {Month:"January",Profit:2000,Expenses:1500,Amount:450},
        {Month:"February",Profit:1000,Expenses:200,Amount:600},
        {Month:"March",Profit:1500,Expenses:500,Amount:300},
        {Month:"April",Profit:500,Expenses:300,Amount:500},
        {Month:"May",Profit:1000,Expenses:450,Amount:250},
        {Month:"June",Profit:2000,Expenses:500,Amount:700}
     ];
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also use MXML to define the content of an Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfMXMLObjectsDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
  <fx:Declarations>
      <fx:Array id="expenses">
         <fx:Object
            Month="January"
            Profit="2000"
            Expenses="1500"
            Amount="450"
         />
         <fx:Object
            Month="February"
            Profit="1000"
            Expenses="200"
            Amount="600"
         />
         <fx:Object
            Month="March"
            Profit="1500"
            Expenses="500"
            Amount="300"
         />
         <fx:Object
            Month="April"
            Profit="500"
            Expenses="300"
            Amount="500"
         />
         <fx:Object
```

```
            Month="May"
            Profit="1000"
            Expenses="450"
            Amount="250"
         />
         <fx:Object
            Month="June"
            Profit="2000"
            Expenses="500"
            Amount="700"
         />
      </fx:Array>
  </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

When defining data in an Array in MXML, be sure to wrap it in a `<fx:Declarations>` tag.

You can also define objects in MXML with a more verbose syntax, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfVerboseMXMLObjects.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
  <fx:Declarations>
      <fx:Array id="expenses">
         <fx:Object>
           <fx:Month>January</fx:Month>
           <fx:Profit>2000</fx:Profit>
           <fx:Expenses>1500</fx:Expenses>
           <fx:Amount>450</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>February</fx:Month>
           <fx:Profit>1000</fx:Profit>
           <fx:Expenses>200</fx:Expenses>
           <fx:Amount>600</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>March</fx:Month>
           <fx:Profit>1500</fx:Profit>
           <fx:Expenses>500</fx:Expenses>
           <fx:Amount>300</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>April</fx:Month>
           <fx:Profit>500</fx:Profit>
           <fx:Expenses>300</fx:Expenses>
           <fx:Amount>300</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>May</fx:Month>
           <fx:Profit>1000</fx:Profit>
           <fx:Expenses>450</fx:Expenses>
           <fx:Amount>250</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>June</fx:Month>
           <fx:Profit>2000</fx:Profit>
           <fx:Expenses>500</fx:Expenses>
           <fx:Amount>700</fx:Amount>
         </fx:Object>
      </fx:Array>
  </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
```

```
     <mx:ColumnChart id="myChart" dataProvider="{expenses}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

A disadvantage of using a simple Array as a chart's data provider is that you can use only the methods of the Array class to manipulate the data. In addition, when you use an Array as a data provider, the data in it must be static. Even if you make the Array bindable, when data in an Array changes, the chart does not reflect those changes. For more robust data manipulation and data binding, you can use a collection for the chart data provider, as described in "Using collections as data providers" on page 1096.

Using collections as data providers

Collections are a more robust data provider mechanism than Arrays. They provide operations that include the insertion and deletion of objects as well as sorting and filtering. Collections also support change notification. An ArrayCollection object provides an easy way to expose an Array as an ICollectionView or IList interface.

As with Arrays, you can use MXML to define the contents of a collection, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfMXMLObjectsDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

    <fx:Declarations>
        <s:ArrayCollection id="expenses">
             <fx:Object
                Month="January"
                Profit="2000"
                Expenses="1500"
                Amount="450"/>
             <fx:Object
                Month="February"
                Profit="1000"
                Expenses="200"
                Amount="600"/>
             <fx:Object
                Month="March"
                Profit="1500"
                Expenses="500"
                Amount="300"/>
             <fx:Object
                Month="April"
                Profit="500"
                Expenses="300"
                Amount="500"/>
             <fx:Object
                Month="May"
                Profit="1000"
                Expenses="450"
                Amount="250"/>
             <fx:Object
                Month="June"
                Profit="2000"
                Expenses="500"
                Amount="700"/>
          </s:ArrayCollection>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Or you can define an object in MXML using child tags rather than attributes:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfVerboseMXMLObjects.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <fx:Declarations>
      <s:ArrayCollection id="expenses">
         <fx:Object>
           <fx:Month>January</fx:Month>
           <fx:Profit>2000</fx:Profit>
           <fx:Expenses>1500</fx:Expenses>
           <fx:Amount>450</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>February</fx:Month>
           <fx:Profit>1000</fx:Profit>
           <fx:Expenses>200</fx:Expenses>
           <fx:Amount>600</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>March</fx:Month>
           <fx:Profit>1500</fx:Profit>
           <fx:Expenses>500</fx:Expenses>
           <fx:Amount>300</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>April</fx:Month>
           <fx:Profit>500</fx:Profit>
           <fx:Expenses>300</fx:Expenses>
           <fx:Amount>300</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>May</fx:Month>
           <fx:Profit>1000</fx:Profit>
           <fx:Expenses>450</fx:Expenses>
           <fx:Amount>250</fx:Amount>
         </fx:Object>
         <fx:Object>
           <fx:Month>June</fx:Month>
           <fx:Profit>2000</fx:Profit>
           <fx:Expenses>500</fx:Expenses>
           <fx:Amount>700</fx:Amount>
         </fx:Object>
      </s:ArrayCollection>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can create an ArrayCollection object in ActionScript. If you define an ArrayCollection in this way, ensure that you import the mx.collections.ArrayCollection class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfObjects.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     private var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
  ]]></fx:Script>

  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

If your data is in an Array, you can pass the Array to the ArrayCollection's constructor to convert it to an ArrayCollection. The following example creates an Array, and then converts it to an ArrayCollection:

```
<?xml version="1.0"?>
<!-- charts/ArrayConvertedToArrayCollection.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     private var expenses:Array = [
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ];
     [Bindable]
     public var expensesAC:ArrayCollection =
        new ArrayCollection(expenses);
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expensesAC}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expensesAC}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Similarly, you can use an `<s:ArrayCollection>` tag to perform the conversion:

```
<?xml version="1.0"?>
<!-- charts/ArrayConvertedToArrayCollectionMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     private var expenses:Array = [
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ];
  ]]></fx:Script>
  <fx:Declarations>
      <s:ArrayCollection id="expensesAC" source="{expenses}"/>
  </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expensesAC}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expensesAC}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The data in ArrayCollections can be bound to the chart's data provider so that the data can be updated in real-time. The following example creates an object with elapsed time and total memory usage every second. It then pushes that new object onto an ArrayCollection that is used as the data provider for a line chart. As a result, the chart itself updates every second showing memory usage of Adobe® Flash® Player or Adobe® AIR™ over time.

```
<?xml version="1.0"?>
<!-- charts/RealTimeArrayCollection.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initTimer()"
    height="600">

  <fx:Script><![CDATA[
      import flash.utils.Timer;
      import flash.events.TimerEvent;
      import mx.collections.ArrayCollection;
      [Bindable]
      public var memoryUsage:ArrayCollection = new ArrayCollection();
      public function initTimer():void {
         // The first parameter in the Timer constructor
         // is the interval, in milliseconds.
         // The second parameter is how many times to run (0 is
         // infinity).
         var myTimer:Timer = new Timer(1000, 0);
         // Add the listener for the timer event.
         myTimer.addEventListener("timer", timerHandler);
         myTimer.start();
      }
      public function timerHandler(event:TimerEvent):void {
         var o:Object = new Object();
         // Get the number of milliseconds since Flash Player or AIR started.
         o.time = getTimer();
         // Get the total memory Flash Player or AIR is using.
         o.memory = flash.system.System.totalMemory;
         trace(o.time + ":" + o.memory);
         // Add new object to the ArrayCollection, which is bound
         // to the chart's data provider.
         memoryUsage.addItem(o);
      }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Memory Usage">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
      <mx:LineChart id="chart"
        dataProvider="{memoryUsage}"
         showDataTips="true">
         <mx:horizontalAxis>
            <mx:LinearAxis/>
         </mx:horizontalAxis>
         <mx:verticalAxis>
            <mx:LinearAxis minimum="5000000"/>
         </mx:verticalAxis>
         <mx:series>
            <mx:LineSeries yField="memory"/>
         </mx:series>
      </mx:LineChart>
    </s:Panel>
</s:Application>
```

Data collections can be paged, which means that data is sent to the client in chunks as the application requests it. But Flex charting controls display all of the data all of the time, by default. As a result, when you use data collections with charts, you should disable the paging features or use non-paged views of the data collection for chart data.

For more information on using collections, see “Data providers and collections” on page 898.

Using an XML file as a data provider

You can define data provider data in a structured file. The following example shows the contents of the data.xml file:

```
<data>
    <result month="Jan-04">
        <apple>81768</apple>
        <orange>60310</orange>
        <banana>43357</banana>
    </result>
    <result month="Feb-04">
        <apple>81156</apple>
        <orange>58883</orange>
        <banana>49280</banana>
    </result>
</data>
```

You can load the file directly as a source of a Model, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/XMLFileDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <fx:Declarations>
        <fx:Model id="results" source="../assets/data.xml"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{results.result}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="banana" displayName="Banana"/>
           <mx:LineSeries yField="apple" displayName="Apple"/>
           <mx:LineSeries yField="orange" displayName="Orange"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can use more complex XML to define the data provider's data. For example, an XML-based data provider can have nested tags. In that case, however, you must use a `dataFunction` to define the fields that the chart uses. For more information, see "Structure of chart data" on page 1113.

To use an ArrayCollection as the chart's data provider, you convert the Model to an ArrayCollection, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/XMLFileToArrayCollectionDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

    <fx:Declarations>
        <fx:Model id="results" source="../assets/data.xml"/>
        <s:ArrayCollection id="myAC"
            source="{ArrayUtil.toArray(results.result)}"/>
    </fx:Declarations>
    <fx:Script>
        import mx.utils.ArrayUtil;
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{myAC}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="banana" displayName="Banana"/>
           <mx:LineSeries yField="apple" displayName="Apple"/>
           <mx:LineSeries yField="orange" displayName="Orange"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Using HTTPService as a data provider

You can define an XML file or any page that returns an XML result as a URL for an HTTPService component. You then bind the HTTPService result directly to the chart’s data provider, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
          <mx:LineSeries
                yField="profit"
                displayName="Profit"/>
           <mx:LineSeries
                yField="expenses"
                displayName="Expenses"/>
         </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

As with most other non-visual components, be sure to wrap the `<mx:HTTPService>` tag in a `<fx:Declarations>` tag.

In this example, the target URL returns the following XML:

```
<data>
    <result month="Jan">
        <profit>2000</profit>
        <expenses>1500</expenses>
        <amount>450</amount>
    </result>
    <result month="Feb">
        <profit>1000</profit>
        <expenses>200</expenses>
        <amount>600</amount>
    </result>
    <result month="Mar">
        <profit>1500</profit>
        <expenses>500</expenses>
        <amount>300</amount>
    </result>
</data>
```

To use an ArrayCollection, you cast the HTTPService result, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceToArrayCollectionDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <mx:HTTPService id="srv"
            url="http://aspexamples.adobe.com/chart_examples/expenses-xml.aspx"
            useProxy="false"
            result="myData=ArrayCollection(srv.lastResult.data.result)"
        />
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var myData:ArrayCollection;
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
```

```
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{myData}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
          <mx:LineSeries
                yField="profit"
                displayName="Profit"/>
           <mx:LineSeries
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also set the result format of the HTTPService to E4X, and then use it as a source for an XMLListCollection object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceToXMLListCollection.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <mx:HTTPService id="srv"
            url="http://aspexamples.adobe.com/chart_examples/expenses-xml.aspx"
            resultFormat="e4x"
        />
        <mx:XMLListCollection id="myAC"
            source="{srv.lastResult.result}"
        />
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.utils.ArrayUtil;
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
```

```
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{myAC}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="@month"/>
        </mx:horizontalAxis>
        <mx:series>
          <mx:LineSeries
                yField="profit"
                displayName="Profit"/>
           <mx:LineSeries
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Another approach is to define the target URL of the HTTPService component as an HTML page that contains data. For example, you might have an HTML page with source that looks like the following:

```
<html>
<body>
<h1>This is a data page</h1>
<!-- data starts here -->
<entry>
    <month>Jan</month>
    <misc>15300</misc>
</entry>
<entry>
    <month>Feb</month>
    <misc>17800</misc>
</entry>
<entry>
    <month>Mar</month>
    <misc>20000</misc>
</entry>
<!-- data ends here -->
</body>
</html>
```

In your application, you can extract the data from the HTML page by using regular expressions. You can then assign the resulting string to an XMLList and use it as a data provider, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTMLDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()"
    height="600">
    <fx:Declarations>
        <mx:HTTPService id="serviceInput"
            resultFormat="text"
            url="http://aspexamples.adobe.com/chart_examples/testDataPage.html"
        />
    </fx:Declarations>

    <fx:Script><![CDATA[
        import mx.events.FlexEvent;
        import mx.rpc.events.ResultEvent;
        import mx.rpc.events.FaultEvent;

        [Bindable]
        public var chartData:XMLList;
        private function initApp():void {
            serviceInput.addEventListener(ResultEvent.RESULT, resultHandler);
            serviceInput.addEventListener(FaultEvent.FAULT, faultHandler);
            serviceInput.send();
        }
        public function resultHandler(evt:Event):void {
            // Extract the XML data from the HTML page.
            var htmlBlob:String = serviceInput.lastResult.toString();
            htmlBlob = htmlBlob.slice( htmlBlob.indexOf("<!-- data starts here -->", 0) +
                25, htmlBlob.length );
        htmlBlob = htmlBlob.slice( 0, htmlBlob.indexOf("<!-- data ends here -->", 0) );
            // Create an XMLList from the extracted XML data.
            chartData = new XMLList(htmlBlob);
        }
        public function faultHandler(evt:Event):void {
            trace(evt.toString());
        }
    ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line chart with data extracted from HTML page">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
         <mx:LineChart id="chart"
            dataProvider="{chartData}" showDataTips="true">
             <mx:series>
                 <mx:LineSeries id="series1" yField="misc" xField="month"/>
             </mx:series>
             <mx:horizontalAxis>
                 <mx:CategoryAxis categoryField="month"/>
             </mx:horizontalAxis>
         </mx:LineChart>
     </s:Panel>
</s:Application>
```

To load the data from a URL on another domain, that target domain must have a crossdomain.xml policy file in place. That policy file must allow the domain on which your Flex application is running to access its data.

In this example, if you requested the data page, the data does not render in the brower because it is wrapped in tags that are outside of the HTML specification. If the data were meant to be displayed, it might be surrounded in HTML entities for the left and right angle brackets, like the following example shows:

```
<html>
<body>
<h1>This is a data page</h1>
<!-- data starts here -->
&lt;entry&gt;
    &lt;month&gt;Jan&lt;/month&gt;
    &lt;misc&gt;15300&lt;/misc&gt;
&lt;/entry&gt;
&lt;entry&gt;
    &lt;month&gt;Feb&lt;/month&gt;
    &lt;misc&gt;17800&lt;/misc&gt;
&lt;/entry&gt;
&lt;entry&gt;
    &lt;month&gt;Mar&lt;/month&gt;
    &lt;misc&gt;20000&lt;/misc&gt;
&lt;/entry>
<!-- data ends here -->
</body>
</html>
```

In that case, you could replace the HTML entities with angle brackets after the source was loaded, as the following example shows:

```
var tagPattern:RegExp = /<[^<>]*>/g;
htmlBlob = htmlBlob.replace(tagPattern, "");

var ltPattern:RegExp = /&lt;/g;
htmlBlob = htmlBlob.replace(ltPattern, "<");

var gtPattern:RegExp = /&gt;/g;
htmlBlob = htmlBlob.replace(gtPattern, ">");
```

Randomly generating chart data

A useful way to create data for use in sample charts is to generate random data. The following example generates test data for use with the chart controls:

```
<?xml version="1.0"?>
<!-- charts/RandomDataGeneration.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.*;
     // Define data provider array for the chart data.
     [Bindable]
     public var dataSet:ArrayCollection;
     // Define the number of elements in the array.
     public var dsLength:Number = 10;
     public function initApp():void {
        // Initialize data provider array.
        dataSet = new ArrayCollection(genData());
     }
     public function genData():Array {
        var result:Array = [];
        for (var i:int=0;i<dsLength;i++) {
           var localVals:Object = {
                valueA:Math.random()*100,
                valueB:Math.random()*100,
                valueX:Math.random()*100,
                valueY:Math.random()*100
           };
           // Push new object onto the data array.
           result.push(localVals);
        }
        return result;
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PlotChart id="myChart"
        dataProvider="{dataSet}" showDataTips="true">
        <mx:series>
           <mx:PlotSeries
                xField="valueX"
                yField="valueY"
                displayName="Series 1"/>
           <mx:PlotSeries
                xField="valueA"
                yField="valueB"
                displayName="Series 2"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend id="l1" dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Structure of chart data

In most cases, the data that is used as the chart's data provider is made up of scalar values. For example, objects contain a single set of fields:

```
{month: "Aug", close: 45.87, open:25.19},
```

Or XML data contains a single set of child tags in a flat structure:

```
<stock>
    <month>Aug</month>
    <close>45.87</close>
    <open>25.19</open>
</stock>
```

In these cases, you assign the data provider's fields to items in the chart by using the `xField` and `yField` for the series, or the `categoryField` for the axis; for example:

```
<mx:ColumnSeries yField="close"/>
<mx:CategoryAxis categoryField="month"/>
```

However, the structure of the chart data can be made up of more complex objects, such as objects within objects or XML with nested child tags. For example, you can embed an object within an object:

```
{month: "Aug", close: {High:45.87,Low:12.2}, open:25.19}
```

Or use nested tags in an XML object:

```
<stock>
    <date>
        <month>Aug</month>
    </date>
    <price>
        <close>45.87</close>
        <open>25.19</open>
    </price>
</stock>
```

In these cases, you cannot refer to the target data by using the flat naming such as `yField="close"`. You also cannot use dot notation. For example, `yField="values.close"` or `categoryField="data.month"` are not valid. Instead, you must use a `dataFunction` method to define which ChartItem fields are populated by the data provider.

For the CategoryAxis class, the `dataFunction` has the following signature:

```
function_name(axis:AxisBase, item:Object):Object
```

Where *axis* is the base class for the current axis, and *item* is the item in the data provider.

For the Series class, the `dataFunction` has the following signature:

```
function_name(series:Series, item:Object, fieldName:String):Object
```

Where *series* is a reference to the current series, *item* is the item in the data provider, and *fieldName* is the field in the current ChartItem that will be populated.

The following example creates two functions that access complex data for both the axis and the series:

```
<?xml version="1.0"?>
<!-- charts/DataFunctionExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
  <fx:Script><![CDATA[
    import mx.charts.chartClasses.AxisBase;
    import mx.charts.chartClasses.Series;
    import mx.charts.CategoryAxis;
    import mx.charts.chartClasses.IAxis;
    import mx.charts.chartClasses.ChartBase;
    import mx.charts.chartClasses.CartesianTransform;

    // This data provider contains complex, nested objects.
    [Bindable]
    public var SMITH:Array = [
        {month: "Aug", close: {High:45.87,Low:12.2}, open:25.19},
        {month: "Sep", close: {High:45.74,Low:10.23}, open:35.29},
        {month: "Oct", close: {High:45.77,Low:12.13}, open:45.19},
        {month: "Nov", close: {High:46.06,Low:10.45}, open:15.59},
    ];
    private function dataFunc(series:Series, item:Object, fieldName:String):Object {
        trace("fieldName: " + fieldName);
        if(fieldName == "yValue" && series.id=="highClose")
            return(item.close.High);
        else if(fieldName == "yValue" && series.id=="lowClose")
            return(item.close.Low);
        else if(fieldName == "xValue")
            return(item.month);
        else
            return null;
      }
    private function catFunc(axis:AxisBase, item:Object):Object {
        for (var s:String in item) {
            trace(s + ":" + item[s]);
        }

        return(item.month);
    }
   ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="ColumnChart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="chart"
            dataProvider="{SMITH}"
            showDataTips="true"
            width="100%"
            height="100%">
            <mx:horizontalAxis>
                <!-- The dataFunction replaces "categoryField='month'. -->
                <mx:CategoryAxis id="h1" dataFunction="catFunc"/>
            </mx:horizontalAxis>
            <mx:series>
                <!-- The dataFunction replaces yField value, which cannot drill
                down into an object (close.High is not valid). -->
                <mx:ColumnSeries id="highClose"
                    displayName="Close (High)"
                    dataFunction="dataFunc"/>
                <!-- The dataFunction replaces yField value, which cannot drill
                down into an object (close.Low is not valid). -->
                <mx:ColumnSeries id="lowClose"
                    displayName="Close (Low)"
                    dataFunction="dataFunc"/>
            </mx:series>
        </mx:ColumnChart>
    </s:Panel>
</s:Application>
```

Changing chart data at run time

Using ActionScript, you can change a charting control's data at run time by using a variety of methods.

You can change a chart or a series data provider. To do this with an HTTPService as the source of your data provider, you change the URL and then call the HTTPService control's `send()` method again.

The following example changes the data provider when the user selects a new ticker symbol from the ComboBox control:

```
<?xml version="1.0"?>
<!-- charts/ChangeSrvDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
    <fx:Script><![CDATA[
        public function changeDataProvider(event:Event):void {
            srv.url = "http://aspexamples.adobe.com/chart_examples/stocks-xml.aspx" +
                "?" + "tickerSymbol=" + event.currentTarget.selectedItem;
                srv.send();
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}" 
showDataTips="true">
            <mx:horizontalAxis>
                <mx:DateTimeAxis dataUnits="days"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries 
                    yField="close" xField="date" 
                    displayName="Closing Price of {cb1.selectedItem}"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
    <s:ComboBox id="cb1" change="changeDataProvider(event)">
        <s:ArrayCollection>
            <fx:String>FRED</fx:String>
            <fx:String>STRK</fx:String>
            <fx:String>FE</fx:String>
        </s:ArrayCollection>
    </s:ComboBox>
</s:Application>
```

The following example defines the data providers as ArrayCollections. It then binds the data provider to a local variable. The example then toggles the chart’s data provider using that local variable when the user clicks the button.

```
<?xml version="1.0"?>
<!-- charts/ChangeDataProvider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <fx:Script>
        <![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
            {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
            {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
        ]);
        [Bindable]
        public var expenses2:ArrayCollection = new ArrayCollection([
            {Month:"Jan", Profit:2400, Expenses:1509, Amount:950},
            {Month:"Feb", Profit:3000, Expenses:2200, Amount:400},
            {Month:"Mar", Profit:3500, Expenses:1200, Amount:200}
        ]);
        [Bindable]
        public var dp:ArrayCollection = expenses;
        public function changeDataProvider():void {
            if (dp==expenses) {
                dp = expenses2;
            } else {
                dp = expenses;
            }
        }
        ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Line Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:LineChart id="myChart" dataProvider="{dp}" showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis
                    dataProvider="{dp}"
                    categoryField="Month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:LineSeries
                    yField="Profit"
                    displayName="Profit"/>
                <mx:LineSeries
                    yField="Expenses"
                    displayName="Expenses"/>
                <mx:LineSeries
                    yField="Amount"
                    displayName="Amount"/>
            </mx:series>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
    <s:Button id="b1"
        label="Change Data Provider"
        click="changeDataProvider()"/>
</s:Application>
```

You can add or remove a data point from a series in ActionScript. The following example adds an item to the existing data provider when the user clicks the button:

```
<?xml version="1.0"?>
<!-- charts/AddDataItem.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns="*" height="600" width="700">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var dpac:ArrayCollection =  new ArrayCollection([
           {A:2000},
           {A:3000},
           {A:4000},
           {A:4000},
           {A:3000},
           {A:2000},
           {A:6000}
        ]);

        public function addDataItem():void {
            var o:Object = {"A":2000};
            dpac.addItem(o);
        }

        private function removeItem():void {
            var i:int = dpac.length;
            if (i > 0) {
                dpac.removeItemAt(i - 1);
            }
        }
    ]]></fx:Script>

  <s:Panel title="Column Chart">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        height="400"
        width="600"
        dataProvider="{dpac}">
        <mx:series>
            <mx:ColumnSeries yField="A" displayName="Series 1"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
  <s:HGroup>
      <s:Button label="Add Data Item" click="addDataItem();"/>
      <s:Button label="Remove Data Item" click="removeItem();"/>
  </s:HGroup>
</s:Application>
```

You can also change the fields of a series to change chart data at run time. You do this by changing the value of the data provider field (such as `xField` or `yField`) on the series object. To get a reference to the series, you use the series' `id` property or the chart control's series index. The following example toggles the data series when the user clicks on the Change Series button:

```
<?xml version="1.0"?>
<!-- charts/ToggleSeries.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initApp();"
    height="600">
  <fx:Script><![CDATA[
     [Bindable]
     public var dataSet:Array;
     public var myStates:Array =
        ["Wisconsin","Ohio","Arizona","Penn"];
     public var curSeries:String;
     public function initApp():void {
        var newData:Array = [];
        for(var i:int=0;i<myStates.length;i++) {
           newData[i] = {
             Apples: Math.floor(Math.random()*150),
             Oranges: Math.floor(Math.random()*150),
             myState: myStates[i]
           }
        }
        dataSet = newData;
        curSeries = "apples";
     }
     public function changeData():void {
        var series:Object = myChart.series[0];
        if (curSeries == "apples") {
           curSeries="oranges";
           series.yField = "Oranges";
           series.displayName = "Oranges";
           series.setStyle("fill",0xFF9933);
        } else {
           curSeries="apples";
           series.yField = "Apples";
           series.displayName = "Apples";
           series.setStyle("fill",0xFF0000);
        }
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart"
            dataProvider="{dataSet}" showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis dataProvider="{dataSet}"
                    categoryField="myState"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries yField="Apples" displayName="Apples">
                    <mx:fill>
                        <mx:SolidColor color="0xFF0000"/>
                    </mx:fill>
                </mx:ColumnSeries>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
    <s:Button id="b1" label="Toggle Series" click="changeData()"/>
</s:Application>
```

You can take advantage of data binding to make your chart reflect data changes in real time. The following example uses a Timer to define the intervals at which it checks for new data. Because the chart's data provider is bound to an ArrayCollection, whenever a new data point is added to the collection, the chart is automatically updated.

```
<?xml version="1.0"?>
<!-- charts/WatchingCollections.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initData();"
    height="600">
    <fx:Declarations>
        <mx:SeriesInterpolate id="interp"
            elementOffset="0"
            duration="300"
            minimumElementDuration="0"
        />
    </fx:Declarations>

    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var dataSet:ArrayCollection;
        [Bindable]
        public var revenue:Number = 100;
        private var t:Timer;
        private function initData():void {
            dataSet = new ArrayCollection();
            t = new Timer(500);
        }
        private function startApp():void {
            t.addEventListener(TimerEvent.TIMER, addData);
            t.start();
        }
        private function addData(e:Event):void {
            /* Add a maximum of 100 data points before user has to click
               the Start button again. */
            if (dataSet.length > 100) {
                stopApp();
            }
            dataSet.addItem( { revenue: revenue } );
            revenue += Math.random() * 10 - 5;
        }

        private function stopApp():void {
            t.stop();
            t.removeEventListener(TimerEvent.TIMER, addData);
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Line Chart">
         <s:layout>
```

```
             <s:HorizontalLayout/>
         </s:layout>
        <mx:LineChart id="myChart" dataProvider="{dataSet}">
            <mx:series>
                <mx:LineSeries
                    yField="revenue"
                    showDataEffect="{interp}"
                    displayName="Revenue"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:LinearAxis autoAdjust="false"/>
            </mx:horizontalAxis>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
    <s:HGroup>
        <s:Button id="b1" label="Start" click="startApp()"/>
        <s:Button id="b2" label="Stop" click="stopApp()"/>
    </s:HGroup>
</s:Application>
```

You can also use system values to update charts at run time. The following example tracks the value of the `totalMemory` property in a LineChart control in real time:

```
<?xml version="1.0"?>
<!-- charts/MemoryGraph.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initTimer()"
    height="600">
  <fx:Script><![CDATA[
      import flash.utils.Timer;
      import flash.events.TimerEvent;
      import mx.collections.ArrayCollection;
      [Bindable]
      public var memoryUsage:ArrayCollection = new ArrayCollection();
        public function initTimer():void {
           /* The first parameter in the Timer constructor
              is the interval, in milliseconds. The second
              parameter is how many times to run (0 is
              infinity). */
           var myTimer:Timer = new Timer(1000, 0);
           /* Add the listener for the timer event. */
           myTimer.addEventListener("timer", timerHandler);
           myTimer.start();
        }
        public function timerHandler(event:TimerEvent):void {
           var o:Object = new Object();
           /* Get the number of milliseconds since Flash
              Player started. */
           o.time = getTimer();
           /* Get the total memory Flash Player is using. */
           o.memory = flash.system.System.totalMemory;
           /* Add new object to the ArrayCollection, which
              is bound to the chart's data provider. */
```

```
                memoryUsage.addItem(o);
            }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="p1" title="Memory Usage">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>

        <mx:LineChart id="chart"
            dataProvider="{memoryUsage}"
            showDataTips="true">
            <mx:horizontalAxis>
                <mx:LinearAxis/>
            </mx:horizontalAxis>
            <mx:verticalAxis>
                <mx:LinearAxis minimum="5000000"/>
            </mx:verticalAxis>
            <mx:series>
                <fx:Array>
                    <mx:LineSeries yField="memory"/>
                </fx:Array>
            </mx:series>
        </mx:LineChart>
    </s:Panel>
</s:Application>
```

Using charts in MX-only applications

An MX-only application is an application that does not use any classes from the spark.swc file in its library path. The main application's root tag is the MX Application class, and no components from the Spark component set are used.

You can use the Flex charting controls in MX-only applications. However, some charting components such as the BarChart and ColumnChart controls, use Spark components by default. As a result, to use charts in MX-only applications, you must apply the MXCharts.css theme to your application. This convenience theme file replaces the use of the Spark Label classes with the MX Label class for the `labelClass` style property.

To apply the MXCharts.css theme to your application on the command line, use the theme compiler option, as the following example shows:

```
mxmlc -theme+=../frameworks/projects/charts/MXCharts.css
```

In Flash Builder, to apply the MXCharts.css theme, use the Additional Compiler Arguments field on the Flex Compiler properties dialog box.

If you set the `compatibility-version` compiler option to 4.0.0, Flex applies the MXCharts.css theme by default.

Chart types

Adobe Flex provides different types of charting controls.

Using area charts

You use the AreaChart control to represent data as an area bounded by a line connecting the data values. The area underneath the line is filled in with a color or pattern. You can use an icon or symbol to represent each data point along the line, or you can show a simple line without icons.

The following example creates an AreaChart control:

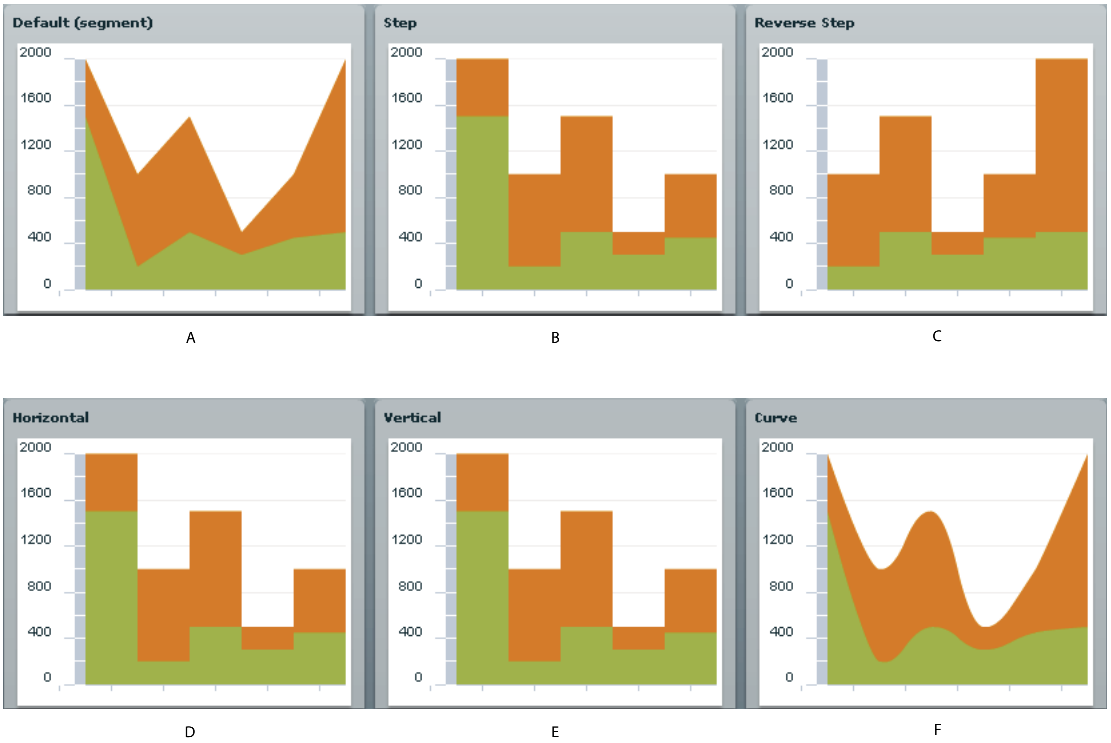
```
<?xml version="1.0"?>
<!-- charts/BasicArea.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Area Chart">
      <s:layout>
          <s:VerticalLayout/>
      </s:layout>
     <mx:AreaChart id="myChart"
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
               categoryField="month"
               displayName="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries yField="profit" displayName="Profit"/>
           <mx:AreaSeries yField="expenses" displayName="Expenses"/>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

An area chart is essentially a line chart with the area underneath the line filled in; therefore, area charts and line charts share many of the same characteristics. For more information, see “Using line charts” on page 1144.

You use the AreaSeries chart series with the AreaChart control to define the data for the chart. The following table describes properties of the AreaSeries chart series that you commonly use to define your chart:

Property	**Description**
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point. If you omit this field, Flex arranges the data points in the order of the data in the data provider.
`minField`	Specifies the field of the data provider that determines the y-axis location of the bottom of an area. This property is optional. If you omit it, the bottom of the area is aligned with the x-axis. This property has no effect on overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see "Using the minField property" on page 1306.
`form`	Specifies the way in which the data series is shown in the chart. The following values are valid: • `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default. • `step` Draws lines as horizontal and vertical segments. At the first data point, draws a horizontal line, and then a vertical line to the second point. Repeats this action for each data point. • `reverseStep` Draws lines as horizontal and vertical segments. At the first data point, draws a vertical line, and then a horizontal line to the second point. Repeats this action for each data point. • `vertical` Draws only the vertical line from the *y*-coordinate of the first point to the *y*-coordinate of the second point at the *x*-coordinate of the second point. Repeats this action for each data point. • `horizontal` Draws only the horizontal line from the *x*-coordinate of the first point to the x-coordinate of the second point at the *y*-coordinate of the first point. Repeats this action for each data point. • `curve` Draws curves between data points.

The following example shows the available forms for an AreaChart control's series:



***A.** Segment (default)* ***B.** Step* ***C.** Reverse Step* ***D.** Horizontal* ***E.** Vertical* ***F.** Curve*

You can use the `type` property of the AreaChart control to represent a number of chart variations, including overlaid, stacked, 100% stacked, and high-low areas. For more information, see “Stacking charts” on page 1308.

To customize the fills of the series in an AreaChart control, you use the `areaFill` and `areaStroke` properties. For each fill, you specify a SolidColor and a SolidColorStroke object. The following example defines three custom SolidColor objects and three custom SolidColorStroke objects, and applies them to the three AreaSeries objects in the AreaChart control.

```
<?xml version="1.0"?>
<!-- charts/AreaChartFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <!-- Define custom colors for use as fills in the AreaChart control. -->
        <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
        <mx:SolidColor id="sc2" color="red" alpha=".3"/>
        <mx:SolidColor id="sc3" color="green" alpha=".3"/>
        <!-- Define custom SolidColorStrokes. -->
        <mx:SolidColorStroke id="s1" color="blue" weight="2"/>
        <mx:SolidColorStroke id="s2" color="red" weight="2"/>
        <mx:SolidColorStroke id="s3" color="green" weight="2"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="AreaChart Control with Custom Fills Example"
        height="100%" width="100%">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <mx:AreaChart id="Areachart"
            height="100%" width="70%"
            paddingLeft="5" paddingRight="5"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}">
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>
            <mx:series>
```

```
                <mx:AreaSeries
                    yField="profit"
                    displayName="Profit"
                    areaStroke="{s1}"
                    areaFill="{sc1}"/>
                <mx:AreaSeries
                    yField="expenses"
                    displayName="Expenses"
                    areaStroke="{s2}"
                    areaFill="{sc2}"/>
                <mx:AreaSeries
                    yField="amount"
                    displayName="Amount"
                    areaStroke="{s3}"
                    areaFill="{sc3}"/>
            </mx:series>
        </mx:AreaChart>
        <mx:Legend dataProvider="{Areachart}"/>
    </s:Panel>
</s:Application>
```

For more information on using fills, see “Using fills with chart controls” on page 1197. For more information on using the SolidColorStroke class, see “Using strokes with chart controls” on page 1192.

Using bar charts

You use the BarChart control to represent data as a series of horizontal bars whose length is determined by values in the data. You can use the BarChart control to represent a number of chart variations, including clustered bars, overlaid, stacked, 100% stacked, and high-low areas. For more information, see “Stacking charts” on page 1308.

You use the BarSeries chart series with the BarChart control to define the data for the chart. The following table describes the properties of the BarSeries chart series that you use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of the base of each bar in the chart. If you omit this property, Flex arranges the bars in the order of the data in the data provider.
`xField`	Specifies the field of the data provider that determines the x-axis location of the end of each bar.
`minField`	Specifies the field of the data provider that determines the x-axis location of the base of a bar. This property has no effect in overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see “Using the minField property” on page 1306.

The following example creates a simple BarChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicBar.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="Profit"/>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

A bar chart is essentially a column chart rotated 90 degrees clockwise; therefore, bar charts and column charts share many of the same characteristics. For more information, see “Using column charts” on page 1137.

Using bubble charts

You use the BubbleChart control to represent data with three values for each data point: a value that determines its position along the x-axis, a value that determines its position along the y-axis, and a value that determines the size of the chart symbol, relative to the other data points in the series.

The `<mx:BubbleChart>` tag takes additional properties, `minRadius` and `maxRadius`. These style properties specify the minimum and maximum radii of the chart elements, in pixels. The data point with the largest value will be less than the `maxRadius` and the data point with the smallest value will be larger than the `minRadius` property. For example, if you set the minRadius to 10 and the maxRadius to 50, all data points will have a radius between 10 and 50. The default value for `maxRadius` is 50 pixels. The default value for `minRadius` is 0 pixels. You can also control the `minRadius` and `maxRadius` properties by using properties of the same name on the BubbleSeries class.

You use the BubbleSeries chart series with the BubbleChart control to define the data for the chart. The following table describes the properties of the BubbleSeries chart series that you commonly use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point. This property is required.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point. This property is required.
`radiusField`	Specifies the field of the data provider that determines the radius of each symbol, relative to the other data points in the series. This property is required.

The following example draws a BubbleChart control and sets the maximum radius of bubble elements to 50:

```
<?xml version="1.0"?>
<!-- charts/BasicBubble.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
```

```
  <s:Panel title="Bubble Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BubbleChart id="myChart"
        minRadius="1"
        maxRadius="50"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <mx:BubbleSeries
                xField="profit"
                yField="expenses"
                radiusField="amount"
                displayName="Profit"/>
        </mx:series>
     </mx:BubbleChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To customize the styles of the bubbles in a BubbleChart control, you use the `fill` and `stroke` properties. You pass these a SolidColor and a SolidColorStroke object, respectively. The following example defines a custom SolidColor object and a custom SolidColorStroke object, and applies them to the BubbleSeries object in the BubbleChart control.

```
<?xml version="1.0"?>
<!-- charts/BubbleFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <!-- Define custom color and line style for the bubbles. -->
        <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
        <mx:SolidColorStroke id="stroke1" color="blue" weight="2"/>
    </fx:Declarations>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="BubbleChart Control with Custom Bubble Styles">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:BubbleChart id="myChart" showDataTips="true">
            <mx:series>
                <mx:BubbleSeries
                    dataProvider="{srv.lastResult.data.result}"
                    displayName="series1"
                    xField="profit"
                    yField="expenses"
                    radiusField="amount"
                    fill="{sc1}"
                    stroke="{stroke1}"/>
            </mx:series>
        </mx:BubbleChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

For more information on using fills, see “Using fills with chart controls” on page 1197. For more information on using the SolidColorStroke class, see “Using strokes with chart controls” on page 1192.

Using multiple series in BubbleChart controls

As with other chart controls, you can have multiple series in a BubbleChart control. There is an additional consideration when using multiple series in a BubbleChart control. You must decide whether you want the size of the bubbles in each series to be relative to bubbles in the other series or relative to only the other bubbles in their own series.

For example, you have two series, A and B. Series A has bubbles with radius values of 10, 20, and 30. Series B has bubbles with radius values of 2, 4, and 8. Your BubbleChart control can display bubbles in series A that are all larger than the bubbles in series B. You can also design a BubbleChart control so that the bubbles’ sizes in series A are not relative to bubbles in series B. Flex renders the bubble with a radius 10 in series A and the bubble with a radius of 2 in series B the same size.

If you want the size of the bubbles to be relative to each other in the different series, add both series in the series array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BubbleRelativeSize.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            private var s1:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 10, "r":10 },
                {"x": 40, "y": 5, "r":20 } ,
                {"x": 60, "y": 0, "r":30 }]);

            [Bindable]
            private var s2:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 50, "r":2 },
                {"x": 40, "y": 75, "r":4 },
                {"x": 60, "y": 100, "r":8 } ]);
    ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Bubble Chart (Bubbles relative to other series)">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:BubbleChart id="myChart"
            showDataTips="true">
            <mx:series>
                <mx:BubbleSeries
                    dataProvider="{s1}"
                    displayName="series1"
                    xField="x" yField="y"
                    radiusField="r"/>
                <mx:BubbleSeries
                    dataProvider="{s2}"
                    displayName="series2"
                    xField="x" yField="y"
                    radiusField="r"/>
            </mx:series>
        </mx:BubbleChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

If you want the size of the bubbles to be relative to each other in their own series, but not to other series, use a different radial axis for each series. To do this, you add a `<mx:radialAxis>` child tag to the `<mx:BubbleSeries>` tag in the MXML. This creates two series in the BubbleChart control whose bubble sizes are independent of one another. The following example shows a BubbleChart control with two independant series:

```
<?xml version="1.0"?>
<!-- charts/BubbleNonRelativeSize.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            private var s1:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 10, "r":10 },
                {"x": 40, "y": 5, "r":20 } ,
                {"x": 60, "y": 0, "r":30 }]);

            [Bindable]
            private var s2:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 50, "r":1 },
                {"x": 40, "y": 75, "r":2 },
                {"x": 60, "y": 100, "r":3 } ]);
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Bubble Chart (Bubbles not relative across series)">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:BubbleChart id="myChart"
            showDataTips="true">
            <mx:series>
                <mx:BubbleSeries
                    dataProvider="{s1}"
                    displayName="series1"
                    xField="x"
                    yField="y"
                    radiusField="r">
                    <mx:radiusAxis>
                        <mx:LinearAxis/>
                    </mx:radiusAxis>
                </mx:BubbleSeries>
                <mx:BubbleSeries
                    dataProvider="{s2}"
                    displayName="series2"
                    xField="x"
                    yField="y"
                    radiusField="r">
                    <mx:radiusAxis>
                        <mx:LinearAxis/>
                    </mx:radiusAxis>
                </mx:BubbleSeries>
            </mx:series>
        </mx:BubbleChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

Using candlestick charts

The CandlestickChart control represents financial data as a series of candlesticks representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line in each candlestick represent the high and low values for the data point, while the top and bottom of the filled box represent the opening and closing values. Each candlestick is filled differently depending on whether the closing value for the data point is higher or lower than the opening value.

The CandlestickChart control's CandlestickSeries requires all four data points: high, low, open, and close. If you do not want to use opening value data points, you can use the HighLowOpenClose charts, which do not require a data point that represents the opening value. For more information, see "Using HighLowOpenClose charts" on page 1141.

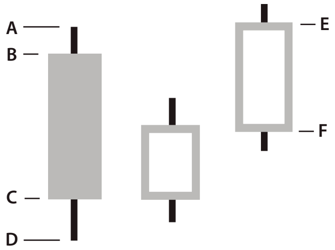
You use the CandlestickSeries chart series with the CandlestickChart control to define the data.

The following table describes the properties of the CandlestickSeries chart series that you commonly use to define your chart:

Property	Description
`closeField`	Specifies the field of the data provider that determines the y-axis location of the closing value of the element. This property defines the top or bottom of the candlestick.
`highField`	Specifies the field of the data provider that determines the y-axis location of the high value of the element. This property defines the top of the line inside the candlestick.
`lowField`	Specifies the field of the data provider that determines the y-axis location of the low value of the element. This property defines the bottom of the line inside the candlestick.
`openField`	Specifies the field of the data provider that determines the y-axis location of the opening value of the element. This property defines the position of the top or bottom of the candlestick.
`xField`	Specifies the field of the data provider that determines the x-axis location of the element. If set to the empty string (""), Flex renders the columns in the order in which they appear in the data provider. The default value is the empty string.

If the `closeField` is lower than the `openField`, Flex applies a solid fill to the candle. The color of this solid fill defaults to the color of the box's outline. It is defined by the `declineFill` style property. If the `closeField` is *higher* than the `openField`, Flex fills the candle with white by default.

The following image shows the properties that define the appearance of the candle. As you can see, the location of the `closeField` property can be either the top or the bottom of the candlestick, depending on whether it is higher or lower than the `openField` property:



A. *highField* ***B.*** *openField* ***C.*** *closeField* ***D.*** *lowField* ***E.*** *closeField* ***F.*** *openField*

The following example creates a CandlestickChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicCandlestick.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Candlestick Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:CandlestickChart id="mychart"
```

```
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        height="400"
        width="400">
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:CandlestickSeries
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
                xField="date"/>
        </mx:series>
     </mx:CandlestickChart>
     <mx:Legend dataProvider="{mychart}"/>
  </s:Panel>
</s:Application>
```

You can change the color of the candle’s fill with the `fill` and `declineFill` properties of the series. The `fill` property defines the color of the candlestick when the `closeField` value is higher than the `openField` value. The `declineFill` property defines the color of the candlestick when the reverse is true. You can also define the properties of the high-low lines and candlestick borders by using the SolidColorStroke class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CandlestickStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Candlestick Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:CandlestickChart id="mychart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        height="400"
        width="400">
        <mx:verticalAxis>
           <mx:LinearAxis title="linear axis" minimum="40" maximum="50"/>
        </mx:verticalAxis>
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:CandlestickSeries
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
                xField="date">
            <mx:fill>
                <mx:SolidColor color="green"/>
            </mx:fill>
            <mx:declineFill>
                <mx:SolidColor color="red"/>
            </mx:declineFill>
            <mx:stroke>
                <mx:SolidColorStroke weight="1" color="black"/>
            </mx:stroke>
           </mx:CandlestickSeries>
        </mx:series>
     </mx:CandlestickChart>
     <mx:Legend dataProvider="{mychart}"/>
  </s:Panel>
</s:Application>
```

Using column charts

The ColumnChart control represents data as a series of vertical columns whose height is determined by values in the data. You can use the ColumnChart control to create several variations of column charts, including simple columns, clustered columns, overlaid, stacked, 100% stacked, and high-low. For more information, see “Stacking charts” on page 1308.

You use the ColumnSeries chart series with the ColumnChart control to define the data for the chart. The following table describes the properties of the ColumnSeries chart series to define your chart:

Property	Description
yField	Specifies the field of the data provider that determines the y-axis location of the top of a column. This field defines the height of the column.
xField	Specifies the field of the data provider that determines the x-axis location of the column. If you omit this property, Flex arranges the columns in the order of the data in the data provider.
minField	Specifies the field of the data provider that determines the y-axis location of the bottom of a column. This property has no effect on overlaid, stacked, or 100% stacked charts. For more information on using the minField property, see “Using the minField property” on page 1306.

The following example creates a ColumnChart control with two series:

```
<?xml version="1.0"?>
<!-- charts/BasicColumn.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To customize the styles of the columns in a ColumnChart control, you specify a SolidColor and a SolidColorStroke object for the `fill` and `stroke` properties, respectively. The following example defines a custom SolidColor object and a custom SolidColorStroke object, and applies them to the ColumnSeries object in the ColumnChart control.

```
<?xml version="1.0"?>
<!-- charts/BasicColumnFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <!-- Define custom colors for use as column fills. -->
        <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
        <mx:SolidColor id="sc2" color="red" alpha=".3"/>
        <!-- Define custom SolidColorStrokes for the columns. -->
        <mx:SolidColorStroke id="s1" color="blue" weight="2"/>
        <mx:SolidColorStroke id="s2" color="red" weight="2"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="ColumnChart Control with Custom Column Styles">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries
                    xField="month"
                    yField="profit"
                    displayName="Profit"
                    fill="{sc1}"
                    stroke="{s1}"/>
                <mx:ColumnSeries
                    xField="month"
                    yField="expenses"
                    displayName="Expenses"
                    fill="{sc2}"
                    stroke="{s2}"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

For more information on using fills, see “Using fills with chart controls” on page 1197. For more information on using the SolidColorStroke class, see “Using strokes with chart controls” on page 1192.

Creating floating column charts

You can also create floating column charts by using the `minField` property of the chart's data series. This property lets you set the lower level of a column.

The following code creates a floating ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/MinFieldColumn.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Revenue:1200, Expenses:500},
        {Month:"Feb", Revenue:1200, Expenses:550},
        {Month:"Mar", Revenue:1240, Expenses:475},
        {Month:"Apr", Revenue:1300, Expenses:600},
        {Month:"May", Revenue:1420, Expenses:575},
        {Month:"Jun", Revenue:1400, Expenses:600},
        {Month:"Jul", Revenue:1500, Expenses:600},
        {Month:"Aug", Revenue:1600, Expenses:750},
        {Month:"Sep", Revenue:1600, Expenses:735},
        {Month:"Oct", Revenue:1750, Expenses:750},
        {Month:"Nov", Revenue:1800, Expenses:800},
        {Month:"Dec", Revenue:2000, Expenses:850}
     ]);
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Floating Column Chart">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <mx:ColumnChart
        dataProvider="{expenses}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Revenue"
                minField="Expenses"
                displayName="Revenue"/>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
</s:Application>
```

For more information, see “Using the minField property” on page 1306.

Using HighLowOpenClose charts

The HLOCChart control represents financial data as a series of lines representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line represent the high and low values for the data point, while the left tick mark represents the opening values and the right tick mark represents the closing values.

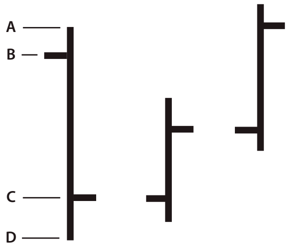
The HLOCChart control does not require a data point that represents the opening value. A related chart is the CandlestickChart control that represents similar data as candlesticks. For more information see “Using candlestick charts” on page 1134.

You use the HLOCSeries with the HLOCChart control to define the data for HighLowOpenClose charts.

The following table describes the properties of the HLOCChart control’s series that you commonly use to define your chart:

Property	**Description**
`closeField`	Specifies the field of the data provider that determines the *y*-axis location of the closing value of the element. This property defines the position of the right tick mark on the vertical line.
`highField`	Specifies the field of the data provider that determines the *y*-axis location of the high value of the element. This property defines the top of the vertical line.
`lowField`	Specifies the field of the data provider that determines the *y*-axis location of the low value of the element. This property defines the bottom of the vertical line.
`openField`	Specifies the field of the data provider that determines the y-axis location of the opening value of the element. This property defines the position of the left tick mark on the vertical line. This property is optional.
`xField`	Specifies the field of the data provider that determines the x-axis location of the element. If set to the empty string (""), Flex renders the columns in the order in which they appear in the data provider. The default value is the empty string.

Data points in an HLOCChart control do not require an `openField` property. If no `openField` property is specified, Flex renders the data point as a flat line with a single closing value indicator pointing to the right. If an `openField` property is specified, Flex renders the data point with another indicator pointing to the left, as the following image shows:



A. highField B. openField C. closeField D. lowField

The following example creates an HLOCChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicHLOC.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="HighLowOpenClose Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:HLOCChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="50"/>
        </mx:verticalAxis>
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:HLOCSeries
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
                xField="date">
           </mx:HLOCSeries>
        </mx:series>
     </mx:HLOCChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can change the stroke of the vertical lines by using a SolidColorStroke object on the series. To change the appearance of the opening and closing value tick marks on the vertical line, you use the `openTickStroke` and `closeTickStroke` style properties. The following example changes the stroke of the vertical line to 2 (the default value is 1) and the color of all the lines to black:

```
<?xml version="1.0"?>
<!-- charts/HLOCStyled.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="HLOC Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:HLOCChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="50"/>
        </mx:verticalAxis>
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:HLOCSeries
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                xField="date"
                displayName="TICKER">
            <mx:stroke>
                <mx:SolidColorStroke color="#000000" weight="2"/>
            </mx:stroke>
            <mx:closeTickStroke>
                <mx:SolidColorStroke color="#000000" weight="1"/>
            </mx:closeTickStroke>
            <mx:openTickStroke>
                <mx:SolidColorStroke color="#000000" weight="1"/>
            </mx:openTickStroke>
           </mx:HLOCSeries>
        </mx:series>
     </mx:HLOCChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

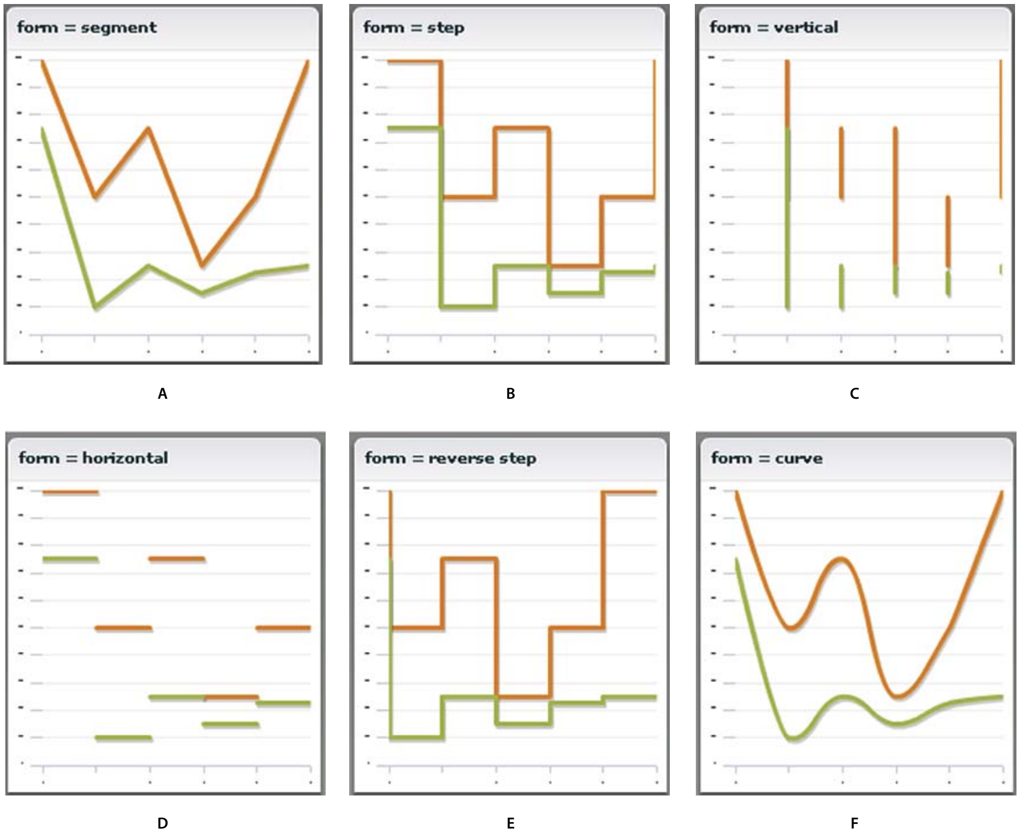
Using line charts

The LineChart control represents data as a series of points, in Cartesian coordinates, connected by a continuous line. You can use an icon or symbol to represent each data point along the line, or show a simple line without icons.

You use the LineSeries chart series with the LineChart control to define the data for the chart. The following table describes the properties of the LineSeries chart series that you commonly use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point. This is the height of the line at that location along the axis.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point. If you omit this field, Flex arranges the data points in the order of the data in the data provider.
`interpolateValues`	Specifies how to represent missing data. If you set the value of this property to `false`, the chart breaks the line at the missing value. If you specify a value of `true`, Flex draws a continuous line by interpolating the missing value. The default value is `false`.
`form`	Specifies the way in which the data series is shown in the chart. The following values are valid: • `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default. • `step` Draws lines as horizontal and vertical segments. At the first data point, draws a horizontal line, and then a vertical line to the second point. Repeats this for each data point. • `reverseStep` Draws lines as horizontal and vertical segments. At the first data point, draws a vertical line, and then a horizontal line to the second point. Repeats this for each data point. • `vertical` Draws the vertical line only from the *y*-coordinate of the first point to the *y*-coordinate of the second point at the *x*-coordinate of the second point. Repeats this for each data point. • `horizontal` Draws the horizontal line only from the *x*-coordinate of the first point to the *x*-coordinate of the second point at the *y*-coordinate of the first point. Repeats this for each data point. • `curve` Draws curves between data points.

The following example shows the available forms for a LineChart control's series:



***A.** segment (default)* ***B.** step* ***C.** vertical* ***D.** horizontal* ***E.** reverseStep* ***F.** curve*

The following example creates a LineChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicLine.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="profit" displayName="Profit"/>
           <mx:LineSeries yField="expenses" displayName="Expenses"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Formatting lines in a LineChart control

You can change the width and color of the lines for each series by using the `<mx:lineStroke>` tag. The default line is 3 pixels wide and has a shadow. The following example sets a custom color and width for the series SolidColorStroke object:

```
<?xml version="1.0"?>
<!-- charts/BasicLineStroke.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart With SolidColorStrokes">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="profit" displayName="Profit">
            <mx:lineStroke>
                <mx:SolidColorStroke 
                    color="0x0099FF" 
                    weight="20" 
                    alpha=".2"/>
            </mx:lineStroke>
           </mx:LineSeries>
           <mx:LineSeries yField="expenses" displayName="Expenses">
            <mx:lineStroke>
                <mx:SolidColorStroke 
                    color="0x0044EB" 
                    weight="20" 
                    alpha=".8"/>
            </mx:lineStroke>
           </mx:LineSeries>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also modify lines in a LineChart with ActionScript. You define a new SolidColorStroke, and apply it to the LineSeries by setting the `lineStroke` style property with the `setStyle()` method. For example:

```
/* First 3 arguments are color, weight, and alpha. */
var s1:SolidColorStroke = new SolidColorStroke(0x0099FF,20,.2);
series1.setStyle("lineStroke", s1);
```

For more information on using the SolidColorStroke class in charts, see “Using strokes with chart controls” on page 1192.

The default appearance of the lines in a LineChart control is with drop shadows. You can remove these shadows by setting the chart control’s `seriesFilters` property to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LineChartNoShadows.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart with No Shadows">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:seriesFilters>
           <fx:Array/>
        </mx:seriesFilters>
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries 
                yField="profit" 
                displayName="Profit"/>
           <mx:LineSeries 
                yField="expenses" 
                displayName="Expenses"/>
           <mx:LineSeries 
                yField="amount" 
                displayName="Amount"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also set the value of the seriesFilters property programmatically, as the following example shows:

```
myLineChart.seriesFilters = [];
```

You can also specify a programmatic renderer (or *skin*) class for each series by setting the lineSegmentRenderer property of the LineSeries. The default renderer is the LineRenderer, but Flex also applies a shadow filter on all series. If you remove the shadow filter, as the previous example shows, but want a line with a drop shadow in your chart, you can set the lineSegmentRenderer to the ShadowLineRenderer class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LineChartOneShadow.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart with One Shadow">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">

        <mx:seriesFilters>
           <fx:Array/>
        </mx:seriesFilters>

        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>

        <mx:series>
           <mx:LineSeries yField="profit" displayName="Profit"/>
           <mx:LineSeries yField="expenses" displayName="Expenses"/>
           <mx:LineSeries yField="amount"
                displayName="Amount"
                lineSegmentRenderer="mx.charts.renderers.ShadowLineRenderer"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

For more information on using renderer classes to change the appearance of ChartItem objects such as the LineChart control's line segments, see "Skinning ChartItem objects" on page 1232.

Using vertical lines in a LineChart control

You can create LineChart controls that show vertical progression.

To make lines in a LineChart control display vertically rather than horizontally, you must do the following:

- Explicitly define the `xField` and `yField` properties for the LineSeries object.
- Set the `sortOnXField` property of the LineSeries object to `false`.

By default, data points in a series are sorted from left to right (on the x-axis) before rendering. This causes the LineSeries to draw horizontally. When you disable the `xField` sort and explicitly define a `yField` property, Flex draws the lines vertically rather than horizontally.

Flex does not sort any data vertically. As a result, you must ensure that your data is arranged correctly in the data provider. If it is not arranged correctly, Flex renders a zig-zagging line up and down the chart as it connects those dots according to position in the data provider.

The following example creates a LineChart control that displays vertical lines:

```
<?xml version="1.0"?>
<!-- charts/VerticalLineChart.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Vertical Line Chart">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:LineSeries
                xField="profit"
                yField="month"
                displayName="Profit"
                sortOnXField="false"/>
           <mx:LineSeries
                xField="expenses"
                yField="month"
                displayName="Expenses"
                sortOnXField="false"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Using pie charts

You use the PieChart control to define a standard pie chart. The data for the data provider determines the size of each wedge in the pie chart relative to the other wedges.

You use the PieSeries chart series with the PieChart control to define the data for the chart. The PieSeries can create standard pie charts or doughnut charts. PieChart controls also support labels that identify data points.

The following table describes the properties of the PieChart control's PieSeries chart series that you commonly use to define your chart:

Property	Description
`field`	Specifies the field of the data provider that determines the data for each wedge of the pie chart.
`labelPosition`	Specifies how to render data labels for the wedges.
`nameField`	Specifies the field of the data provider to use as the name for the wedge in DataTip objects and legends.

The following example defines a PieChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicPie.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Pie Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PieChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <mx:PieSeries
                field="amount"
                nameField="item"
                labelPosition="callout"/>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To customize the colors of the wedges in a PieChart control, you use the `fills` property. You pass this an Array of SolidColor objects. The following example defines an Array of custom SolidColor objects, and applies it to the PieSeries object in the PieChart control.

```
<?xml version="1.0"?>
<!-- charts/PieFilling.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
        <!-- Define custom colors for use as pie wedge fills. -->
        <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
        <mx:SolidColor id="sc2" color="red" alpha=".3"/>
        <mx:SolidColor id="sc3" color="green" alpha=".3"/>
        <mx:SolidColor id="sc4" color="gray" alpha=".3"/>
        <mx:SolidColor id="sc5" color="black" alpha=".3"/>
        <mx:SolidColor id="sc6" color="yellow" alpha=".3"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="PieChart control with custom fills">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
        <mx:PieChart id="pie" 
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:series>
                <mx:PieSeries 
                    field="amount" 
                    nameField="item" 
                    labelPosition="callout"
                    fills="{[sc1, sc2, sc3, sc4, sc5, sc6]}"/>
            </mx:series>
        </mx:PieChart>
        <mx:Legend dataProvider="{pie}"/>
    </s:Panel>
</s:Application>
```

For more information on using fills, see “Using fills with chart controls” on page 1197.

You can also customize the lines that outline each wedge by using the `stroke` property of a PieSeries object. You set this property to an instance of the SolidColorStroke class. For more information on using the SolidColorStroke class, see “Using strokes with chart controls” on page 1192.

Using data labels with PieChart controls

PieChart controls support data labels that display information about each data point. All charts support DataTip objects, which display the value of a data point when the user moves the mouse over it. Data labels, on the other hand, are only supported by PieSeries, ColumnSeries, and BarSeries objects. Data labels are different from DataTip objects in that they are always visible and do not react to mouse movements.

To add data labels to your PieChart control, set the `labelPosition` property on the series to a valid value other than `none`. To remove labels from your pie chart, set the `labelPosition` property to `none`. The default value is `none`.

The following table describes the valid values of the `labelPosition` property for a PieSeries object. The PieSeries class supports more values for this property than the BarSeries and ColumnSeries classes.

Value	Description
`callout`	Draws labels in two vertical stacks on either side of the PieChart control. Shrinks the PieChart if necessary to make room for the labels. Draws key lines from each label to the associated wedge. Shrinks labels as necessary to fit the space provided. This property can only be used with a PieSeries object. You cannot use callouts with BarSeries and ColumnSeries objects.
`inside`	Draws labels inside the chart. Shrinks labels to ensure that they do not overlap each other. Any label that must be drawn too small, as defined by the `insideLabelSizeLimit` property, is hidden from view. When two labels overlap, Flex gives priority to labels for larger slices. not true any more???
`insideWithCallout`	Draws labels inside the pie, but if labels are shrunk below a legible size, Flex converts them to callout labels. You commonly set the value of the `labelPosition` property to `insideWithCallout` when the actual size of your chart is flexible and users might resize it. This property can only be used with a PieSeries object. You cannot use callouts with BarSeries and ColumnSeries objects.
`none`	Does not draw labels. This is the default value.
`outside`	Draws labels around the boundary of the PieChart control.

The following table describes the properties of the PieSeries object that you can use to manipulate the appearance of labels:

Property	Description
`calloutGap`	Defines how much space, in pixels, to insert between the edge of the pie and the data labels when rendering callouts. The default value is 10 pixels.
`calloutStroke`	Defines the line style used to draw the lines to callouts. For more information on defining line data points, see "Using strokes with chart controls" on page 1192.
`insideLabelSizeLimit`	Defines the size threshold, expressed in points, below which inside data labels are considered illegible. Below this threshold, data labels are either removed entirely or turned into callouts based on the setting of the series `labelPosition` property.

You can change the value of the data labels by using the `labelFunction` property of the PieSeries object to specify a callback function. For more information, see "Customizing data label values" on page 1283.

Creating doughnut charts

Flex lets you create doughnut charts out of PieChart controls. Doughnut charts are identical to pie charts, except that they have hollow centers and resemble wheels rather than filled circles.

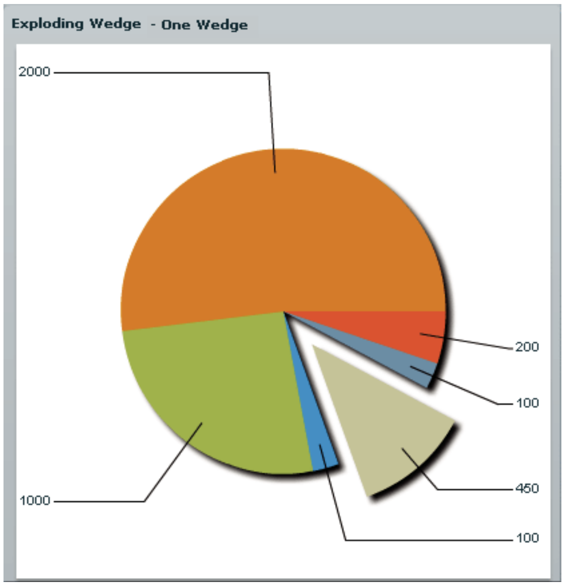
To create a doughnut chart, specify the `innerRadius` property on the PieChart control, as the following example shows:

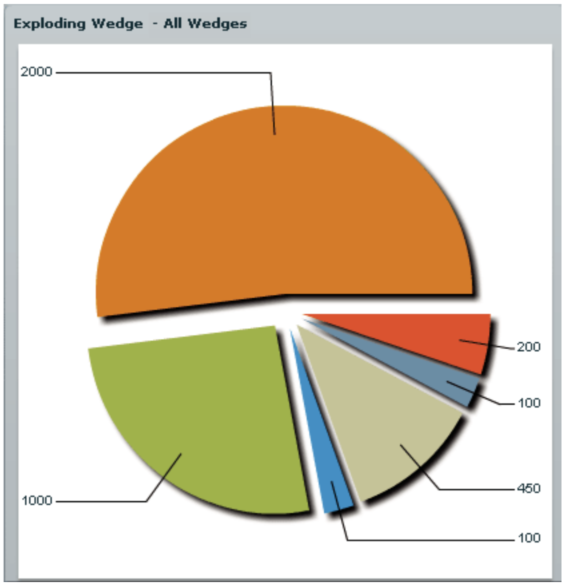
```
<?xml version="1.0"?>
<!-- charts/DoughnutPie.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Pie Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PieChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true"
        innerRadius=".3">
        <mx:series>
           <mx:PieSeries 
                field="amount" 
                nameField="item" 
                labelPosition="callout"/>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The value of the `innerRadius` property is a percentage value of the “hole” compared to the entire pie’s radius. Valid values range from 0 to 1.

Creating exploding pie charts

The PieSeries chart series supports exploding wedges, both uniformly and on a per-wedge basis, so that you can achieve effects similar to the following:





The following table describes the properties that support exploding pie charts:

Property	Description
`explodeRadius`	A value from 0 to 1, representing the percentage of the available pie radius to use when exploding the wedges of the pie.
`perWedgeExplodeRadius`	An array of values from 0 to 1. The Nth value in this array is added to the value of `explodeRadius` to determine the explode amount of each individual wedge of the pie. Individual values can be left undefined, in which case the wedge will only explode according to the `explodeRadius` property.
`reserveExplodeRadius`	A value from 0 to 1, representing an amount of the available pie radius to reserve for animating an exploding wedge.

To explode all wedges of a pie chart evenly, you use the `explodeRadius` property on the PieSeries, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ExplodingPie.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Exploding Pie Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PieChart id="pie"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <!--explodeRadius is a number between 0 and 1.-->
           <mx:PieSeries
                field="amount"
                nameField="item"
                explodeRadius=".12"/>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </s:Panel>
</s:Application>
```

To explode one or more wedges of the pie, you use an Array of `explodeRadius` values. Each value in the Array applies to the corresponding data point. In the following example, the fourth data point, the Car expense, is exploded:

```
<?xml version="1.0"?>
<!-- charts/ExplodingPiePerWedge.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     // Create a bindable Array of explode radii.
     [Bindable]
     public var explodingArray:Array = [0,0,0,.2,0,0]
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Exploding Pie Chart Per Wedge">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PieChart id="pie"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <!--Apply the Array of radii to the PieSeries.-->
           <mx:PieSeries
                field="amount"
                nameField="item"
                perWedgeExplodeRadius="{explodingArray}"
                labelPosition="callout"/>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </s:Panel>
</s:Application>
```

Using plot charts

You use the PlotChart control to represent data in Cartesian coordinates where each data point has one value that determines its position along the x-axis, and one value that determines its position along the y-axis. You can define the shape that Flex displays at each data point with the renderer for the data series.

You use the PlotSeries class with the PlotChart control to define the data for the chart. The following table describes the properties of the PlotSeries chart series that you commonly use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point.
`radius`	Specifies the radius, in pixels, of the symbol at each data point. The default value is `5` pixels.

***Note:** Both the `xField` and `yField` properties are required for each PlotSeries in a PlotChart control.*

The following example defines three data series in a PlotChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicPlot.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PlotChart id="myChart" dataProvider="{srv.lastResult.data.result}"
     showDataTips="true">
        <mx:series>
           <mx:PlotSeries
                xField="expenses"
                yField="profit"
                displayName="Plot 1"/>
           <mx:PlotSeries
                xField="amount"
                yField="expenses"
                displayName="Plot 2"/>
           <mx:PlotSeries
                xField="profit"
                yField="amount"
                displayName="Plot 3"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To customize the styles of the points in a PlotChart control, you define a SolidColor and a SolidColorStroke object for the `fill` and `stroke` properties, respectively. The following example defines three SolidColor objects and three custom SolidColorStroke objects, and applies them to the PlotSeries objects in the PlotChart control.

```
<?xml version="1.0"?>
<!-- charts/PlotFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <!-- Define custom colors for use as plot point fills. -->
        <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
        <mx:SolidColor id="sc2" color="red" alpha=".3"/>
        <mx:SolidColor id="sc3" color="green" alpha=".3"/>
        <!-- Define custom SolidColorStrokes. -->
        <mx:SolidColorStroke id="s1" color="blue" weight="1"/>
        <mx:SolidColorStroke id="s2" color="red" weight="1"/>
        <mx:SolidColorStroke id="s3" color="green" weight="1"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Plot Chart with custom fills">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
        <mx:PlotChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:series>
                <mx:PlotSeries
                    xField="expenses"
                    yField="profit"
                    displayName="Plot 1"
                    fill="{sc1}"
                    stroke="{s1}"/>
                <mx:PlotSeries
                    xField="amount"
                    yField="expenses"
                    displayName="Plot 2"
                    fill="{sc2}"
                    stroke="{s2}"/>
                <mx:PlotSeries
                    xField="profit"
                    yField="amount"
                    displayName="Plot 3"
                    fill="{sc3}"
                    stroke="{s3}"/>
            </mx:series>
        </mx:PlotChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

For more information on using fills, see "Using fills with chart controls" on page 1197. For more information on using the SolidColorStroke class, see "Using strokes with chart controls" on page 1192.

By default, Flex displays the first data series in the chart as a diamond at each point. When you define multiple data series in a chart, Flex rotates the shape for the series (starting with a diamond, then a circle, then a square). If you have more series than there are default renderers, Flex begins again with the diamond.

The diamond shape, like the other shapes, is defined by a renderer class. The renderer classes that define these shapes are in the mx.charts.renderers package. The circle is defined by the CircleItemRenderer class. The following default renderer classes define the appearance of the data points:

- BoxItemRenderer
- CircleItemRenderer
- CrossItemRenderer
- DiamondItemRenderer
- ShadowBoxItemRenderer
- TriangleItemRenderer

You can control the image that is displayed by the chart for each data point by setting the `itemRenderer` style property of the series. The following example overrides the default renderers for the series:

```
<?xml version="1.0"?>
<!-- charts/PlotWithCustomRenderer.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart With Custom Item Renderer">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PlotChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <mx:PlotSeries
                xField="expenses"
                yField="profit"
                displayName="Plot 1"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"
                radius="10"/>
           <mx:PlotSeries
                xField="amount"
                yField="expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.DiamondItemRenderer"
                radius="10"/>
           <mx:PlotSeries
                xField="profit"
                yField="amount"
                displayName="Plot 3"
                itemRenderer=
                "mx.charts.renderers.TriangleItemRenderer"
                radius="10"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also use graphics or custom classes to define each plot point. For more information, see “Creating custom renderers” on page 1235.

Using multiple data series

Charts that are subclasses of the CartesianChart class let you mix different data series in the same chart control. You can create a column chart with a trend line running through it or mix any data series with any other similar series.

You can use any combination of the following series objects in a CartesianChart control:

- AreaSeries
- BarSeries
- BubbleSeries
- CandlestickSeries
- ColumnSeries
- HLOCSeries
- LineSeries
- PlotSeries

The following example mixes a LineSeries and a ColumnSeries:

```
<?xml version="1.0"?>
<!-- charts/MultipleSeries.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_fred.send();srv_strk.send();"
    height="600">
    <fx:Declarations>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
       <!-- View source of the following pages to see the structure of the data that Flex uses
in this example. -->
       <mx:HTTPService id="srv_fred" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FRED"/>
       <mx:HTTPService id="srv_strk" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=STRK"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Multiple Data Series" width="400" height="400">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        height="250"
        width="350">
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
```

```
            <mx:LinearAxis minimum="40" maximum="50"/>
         </mx:verticalAxis>
         <mx:series>
            <mx:ColumnSeries
                 dataProvider="{srv_fred.lastResult.data.result}"
                 xField="date"
                 yField="close"
                 displayName="FRED">
            </mx:ColumnSeries>
            <mx:LineSeries
                 dataProvider="{srv_strk.lastResult.data.result}"
                 xField="date"
                 yField="close"
                 displayName="STRK">
            </mx:LineSeries>
         </mx:series>
      </mx:ColumnChart>
      <mx:Legend dataProvider="{myChart}"/>
   </s:Panel>
</s:Application>
```

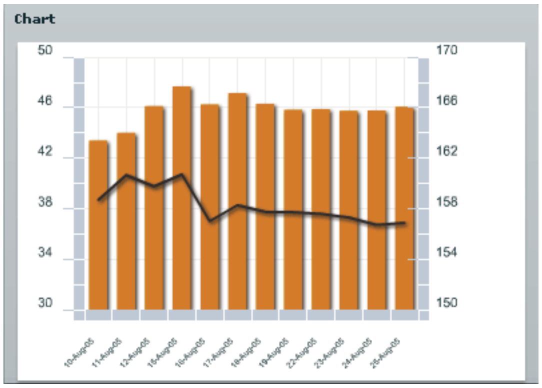
Using multiple series in the same chart works best when the data points are in a similar range (such as a stock price and its moving average). When the data points are in numerically very different ranges, the chart can be difficult to understand because, by default, the data is shown on a single axis. The solution to this problem is to use multiple axes, each with its own range. You can plot each data series on its own axis within the same chart using the techniques described in "Using multiple axes" on page 1164.

Using multiple axes

One potential problem when using more than one data series in a single chart is that if the scales of the data are very different, the data points might be plotted in very different areas on the chart's canvas. For example, one stock price could trade in the range of $100 to $150, while another stock price could fluctuate from $2 to $2.50. If you plot both stocks in the same chart, it would be difficult to see any correlation between the prices, even with a logarithmic axis.

To work around this problem, you can use multiple axes in your charts so that each data series is positioned relative to its own axis. All chart controls that are subclasses of CartesianChart support adding additional sets of data on a additional scales in the horizontal axis, vertical axis, or both. (This applies to all charts except the PieChart control.) You can use values on the additional axes to compare multiple sets of data that are on different scales, such as stock prices that trade in different ranges.

The following example shows a stock price that trades within a $40 to $45 range, and another stock price that trades within a $150 to $160 range. The values of the axis on the left show the range of values of the first stock, and the values of the axis on the right show the range of values of the second stock.



To use multiple axes in a chart, you first define the chart's series and their axes. For example, for a chart that mixes columns with a line, you would have a ColumnSeries and LineSeries. For each of these, you would likely also define the vertical axis, as the following example shows:

```
<mx:series>
    <mx:ColumnSeries id="cs1" dataProvider="{SMITH}" yField="close">
        <mx:verticalAxis>
            <mx:LinearAxis id="v1" minimum="40" maximum="50"/>
        </mx:verticalAxis>
    </mx:ColumnSeries>
    <mx:LineSeries id="cs2" dataProvider="{DECKER}" yField="close">
        <mx:verticalAxis>
            <mx:LinearAxis id="v2" minimum="150" maximum="170"/>
        </mx:verticalAxis>
    </mx:LineSeries>
</mx:series>
```

You then define the axis renderers, and bind their `axis` properties to the series' axes. In this case, you define two vertical axis renderers, and bind them to the LinearAxis objects.

```
<mx:verticalAxisRenderers>
    <mx:AxisRenderer placement="left" axis="{v1}"/>
    <mx:AxisRenderer placement="left" axis="{v2}"/>
</mx:verticalAxisRenderers>
```

Note that you control the location of the axis by using the `placement` property of the AxisRenderer. For vertical axis renderers, valid values are `left` and `right`. For horizontal axis renderers, valid values are `top` and `bottom`.

Axes can be independent of the series definition, too. For example, you can also point more than one series to the same axis. In this case, you could define a horizontal axis, as follows:

```
<mx:horizontalAxis>
    <mx:CategoryAxis id="h1" categoryField="date"/>
<mx:horizontalAxis>
```

And then bind it to the series, like this:

```
<mx:ColumnSeries id="cs1" horizontalAxis="{h1}" dataProvider="{SMITH}" yField="close">
...
<mx:LineSeries id="cs2" horizontalAxis="{h1}" dataProvider="{DECKER}" yField="close">
```

And you can bind an axis renderer to that same axis:

```
<mx:horizontalAxisRenderers>
    <mx:AxisRenderer placement="bottom" axis="{h1}"/>
</mx:horizontalAxisRenderers>
```

The final result is a chart with multiple axes, but whose series share some of the same properties defined by common axis renderers.

```
<?xml version="1.0"?>
<!-- charts/MultipleAxes.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_fe.send();srv_strk.send()"
    height="600">
    <fx:Declarations>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
       <!-- View source of the following pages to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv_fe" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FE"/>
       <mx:HTTPService id="srv_strk" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=STRK"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Multiple Axes with Multiple Data Series"
     width="400" height="400">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        height="250"
        width="350">
        <mx:horizontalAxis>
           <mx:DateTimeAxis id="h1" dataUnits="days"/>
        </mx:horizontalAxis>

          <mx:horizontalAxisRenderers>
            <mx:AxisRenderer placement="bottom" axis="{h1}"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
            <mx:AxisRenderer placement="left" axis="{v1}"/>
            <mx:AxisRenderer placement="right" axis="{v2}"/>
        </mx:verticalAxisRenderers>
        <mx:series>
           <mx:ColumnSeries
                dataProvider="{srv_fe.lastResult.data.result}"
                xField="date"
                yField="close"
                displayName="FE">
                  <mx:verticalAxis>
                     <mx:LinearAxis id="v1" minimum="2" maximum="5"/>
                  </mx:verticalAxis>
           </mx:ColumnSeries>
           <mx:LineSeries
                dataProvider="{srv_strk.lastResult.data.result}"
                xField="date"
                yField="close"
                displayName="STRK">
                  <mx:verticalAxis>
                     <mx:LinearAxis id="v2" minimum="40" maximum="50"/>
                  </mx:verticalAxis>
           </mx:LineSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Even if the `verticalAxisRenderers` or `horizontalAxisRenderers` properties have not been specified, Cartesian charts will create default horizontal and vertical axis renderers based on the default axes of the chart.

When using multiple axes, it is important to recognize that it will not necessarily be immediately apparent which axis applies to which data set in the chart. As a result, you should try to style the axes so that they match the styles of the chart items.

The following example defines two colors and then uses those colors in the axis renderers and in the strokes and fills for the chart items:

```
<?xml version="1.0"?>
<!-- charts/StyledMultipleAxes.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_fe.send();srv_strk.send()"
    height="600">
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var c1:Number = 0x224488;
            [Bindable]
            public var c2:Number = 0x884422;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
       <!-- View source of the following pages to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv_fe" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FE"/>
       <mx:HTTPService id="srv_strk" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=STRK"/>
        <mx:SolidColorStroke id="h1Stroke" color="{c1}"
```

```
            weight="8" alpha=".75" caps="square"/>
        <mx:SolidColorStroke id="h2Stroke"
            color="{c2}" weight="8" alpha=".75" caps="square"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Multiple Axes with Multiple Data Series"
    width="400" height="400">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        height="250" width="350">
        <mx:horizontalAxis>
           <mx:DateTimeAxis id="h1" dataUnits="days"/>
        </mx:horizontalAxis>

          <mx:horizontalAxisRenderers>
            <mx:AxisRenderer placement="bottom" axis="{h1}"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
            <mx:AxisRenderer placement="left" axis="{v1}">
                <mx:axisStroke>{h1Stroke}</mx:axisStroke>
               </mx:AxisRenderer>
            <mx:AxisRenderer placement="right" axis="{v2}">
                <mx:axisStroke>{h2Stroke}</mx:axisStroke>
               </mx:AxisRenderer>
        </mx:verticalAxisRenderers>
        <mx:series>
           <mx:ColumnSeries
                dataProvider="{srv_fe.lastResult.data.result}"
                xField="date" yField="close"
                displayName="FE">
                  <mx:verticalAxis>
                     <mx:LinearAxis id="v1" minimum="2" maximum="5"/>
                  </mx:verticalAxis>
                <mx:fill>
                    <mx:SolidColor color="{c1}"/>
                </mx:fill>
               </mx:ColumnSeries>
           <mx:LineSeries
                dataProvider="{srv_strk.lastResult.data.result}"
                xField="date" yField="close"
                displayName="STRK">
                  <mx:verticalAxis>
                     <mx:LinearAxis id="v2" minimum="40" maximum="50"/>
                  </mx:verticalAxis>
               <mx:lineStroke>
                    <mx:SolidColorStroke color="{c2}" weight="4" alpha="1"/>
                </mx:lineStroke>
               </mx:LineSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can customize labels by using the `labelFunction` property of the AxisRenderer class. This lets you control the labels if you use multiple axes. For more information, see “Customizing axis labels” on page 1270.

For CartesianChart controls, there is no limit to the number of axes you can have.

For PolarChart controls, such as a PieChart, you generally do not use multiple axes because even though each series could have its own angular axis, the angular axis is always from 0 to 360 (the size of the wedge).

Formatting charts

You can customize the look and feel of your Adobe® Flex® charting controls. You can format the appearance of almost all chart elements, from the font properties of an axis label to the width of the stroke in a legend.

To set style properties in your chart controls, you can use Cascading Style Sheets (CSS) syntax or set the style properties inline as tag attributes. As with all styles, you can call the setStyle() method to set any style on your chart elements.

For more information on using the `setStyle()` method, see “Using the setStyle() and getStyle() methods” on page 1540.

Applying chart styles

You can apply style properties to charts by using CSS or inline syntax. You can also apply styles to chart elements by using binding.

Applying styles with CSS

You can apply styles to charting components with CSS definitions. You can set chart properties such as fonts and tick marks, or series properties, such as the fills of the boxes in a ColumnChart.

A limitation of using CSS to style your charts is that the styleable chart properties often use compound values, such as strokes and gradient fills, that cannot be expressed by using CSS. The result is that you cannot express all values of chart styles by using CSS syntax.

To determine if an object's properties can be styled by using CSS, check to see if the class or its parent implements the IStyleClient interface. If it does, then you can set the values of the object's style properties with CSS. If it does not, then the class does not participate in the styles subsystem and you therefore cannot set its properties with CSS. In that case, the properties are public properties of the class and not style properties. You can apply properties with CSS by using pre-defined styles for some classes, such as the `verticalAxisStyleName` and `horizontalAxisStyleName`. For more information, see "Using predefined axis style properties" on page 1174.

Applying CSS to chart controls

You can use the control name in CSS to define styles for that control. This is referred to as a type selector, because the style you define is applied to all controls of that type. For example, the following style definition specifies the font for all BubbleChart controls:

```
<?xml version="1.0"?>
<!-- charts/BubbleStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|BubbleChart {
         fontFamily:Arial;
         fontSize:15;
         color:#FF0033;
      }
    </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
```

```
  <s:Panel title="Bubble Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BubbleChart id="myChart"
        minRadius="1"
        maxRadius="50"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <mx:BubbleSeries
                xField="profit"
                yField="expenses"
                radiusField="amount"
                displayName="Profit"/>
        </mx:series>
     </mx:BubbleChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Some styles, such as `fontSize` and `fontFamily`, are inheritable, which means that if you set them on the chart control, the axes labels, titles, and other text elements on the chart inherit those settings. To determine whether a style is inheritable, see that style's description in *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Axis labels appear next to the tick marks along the chart's axis. Titles appear parallel to the axis line. By default, the text on an axis uses the text styles of the chart control.

Axis elements use whatever styles are set on the chart control's type or class selector, but you can also specify different styles for each axis by using a class selector on the axis renderer or predefined axis style properties.

Applying different styles to each series

To apply different styles to each series in the charts with CSS, you use the `chartSeriesStyles` property. This property takes an Array of Strings. Each String specifies the name of a class selector in the style sheet. Flex applies each of these class selectors to a series.

To apply CSS to a series, you define a type or class selector for the chart that defines the `chartSeriesStyles` property. You then define each class selector named in the `chartSeriesStyles` property.

Essentially, you are defining a new style for each series in your chart. For example, if you have a ColumnChart control with two series, you can apply a different style to each series without having to explicitly set the styles on each series.

The following example defines the colors for two series in the ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/SeriesStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>

  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|ColumnChart {
        chartSeriesStyles: PCCSeries1, PCCSeries2;
     }
     .PCCSeries1 {
        fill: #CCFF66;
     }
     .PCCSeries2 {
        fill: #CCFF99;
     }
  </fx:Style>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <s:Panel title="Setting Styles on Series">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="column"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

Flex sets the `styleName` property of each series to the corresponding selector in the `chartSeriesStyles` Array. You do not have to explicitly set styles for each series. If you have more series than available `chartSeriesStyles` selectors, the chart begins again with the first style.

If you manually set the value of the `styleName` property on a particular series, that style takes priority over the styles that the `chartSeriesStyles` property specifies.

For PieChart controls, you can define the `fills` property of the series. The PieChart applies the values in this Array to the pie wedges. It starts with the first value if there are more wedges than values defined in the Array. The following example creates a PieChart that uses a spectrum of reds for the wedges:

```
<?xml version="1.0"?>
<!-- charts/PieWedgeFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>

  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|PieSeries {
         fills:#FF0000, #FF3333, #FF6666, #FF9999, #FFAAAA, #FFCCCC;
      }
  </fx:Style>
```

```
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <s:Panel title="Setting Pie Wedge Fills with CSS">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PieChart id="pie"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <mx:PieSeries
                field="amount"
                nameField="item"
                labelPosition="callout"/>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </s:Panel>
</s:Application>
```

Using predefined axis style properties

The following predefined class selectors are available for axis styles:

- `horizontalAxisStyleName`
- `verticalAxisStyleName`

Flex applies each style to the corresponding axis.

In addition to these, the following two class selectors define an array of class selectors that define the style properties for the axes:

- `horizontalAxisStyleNames`
- `verticalAxisStyleNames`

In this case, Flex loops through the selectors and applies them to the axis renderers that correspond to their position in the Array.

In addition to these class selectors for the axis style properties, you can use the `axisTitleStyleName` and `gridLinesStyleName` class selectors to apply styles to axis titles and grid lines.

The following example removes tick marks from the horizontal axis:

```
<?xml version="1.0"?>
<!-- charts/PredefinedAxisStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|ColumnChart {
        horizontalAxisStyleName:myAxisStyles;
        verticalAxisStyleName:myAxisStyles;
     }
     .myAxisStyles {
        tickPlacement:none;
     }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Using Predefined Axis Styles">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="column"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
               displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

Using class selectors for axis styles

To use a class selector to define styles for axis elements, define the custom class selector in an `<fx:Style>` block or external style sheet, and then use the `styleName` property of the AxisRenderer class to point to that class selector. Note that any time you want to use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer's `axis` property.

The following example defines the `MyStyle` style and applies that style to the elements on the horizontal axis:

```
<?xml version="1.0"?>
<!-- charts/AxisClassSelectors.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
  <fx:Declarations>
     <!-- View source of the following page to see the structure of the data that Flex uses in
this example. -->
     <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
     <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
  </fx:Declarations>
    <fx:Style>
        .myStyle {
            fontSize:9;
            color:red;
        }
    </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="AxisRenderer Class Selectors example">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="column"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer
```

```
                axis="{a1}"
                styleName="myStyle"/>
        </mx:horizontalAxisRenderers>
        <mx:horizontalAxis>
           <mx:CategoryAxis id="a1" categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

Most charting-specific style properties are not inheritable, which means that if you set the property on a parent object, the child object does not inherit its value.

For more information on using CSS, see “About styles” on page 1492.

Applying styles inline

You can set many styleable elements of a chart as attributes of the MXML tag. For example, to set the styleable properties of an axis, you can use the `<mx:AxisRenderer>` tag rather than define a new style in CSS.

The following example sets the `fontSize` property of the horizontal axis to 7:

```
<mx:horizontalAxisRenderers>
    <mx:AxisRenderer fontSize="7" axis="{h1}"/>
</mx:horizontalAxisRenderers>
```

Notice that any time you want to use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer’s `axis` property.

You can also access the properties of renderers in ActionScript so that you can change their appearance at run time. For additional information about axis renderers, see “Working with axes” on page 1247.

Applying styles by binding tag definitions

You can define styles with MXML tags. You can then bind the values of the renderer properties to those tags. The following example defines the weight and color of the strokes, and then applies those strokes to the chart’s AxisRenderer class:

```
<?xml version="1.0"?>
<!-- charts/BindStyleValues.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
      <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
      <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
      <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
      <mx:SolidColorStroke color="0x00FF00" weight="2" id="axis"/>
      <mx:SolidColorStroke color="0xFF0000" weight="1" id="ticks"/>
      <mx:SolidColorStroke color="0x0000FF" weight="1" id="mticks"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="HLOC Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
      <mx:HLOCChart id="mychart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
         <mx:horizontalAxisRenderers>
            <mx:AxisRenderer
                axis="{a1}"
                axisStroke="{axis}"
                placement="bottom"
                minorTickPlacement="inside"
                minorTickLength="2"
                tickLength="5"
                tickPlacement="inside">
               <mx:tickStroke>{ticks}</mx:tickStroke>
               <mx:minorTickStroke>{mticks}</mx:minorTickStroke>
            </mx:AxisRenderer>
         </mx:horizontalAxisRenderers>
         <mx:verticalAxis>
            <mx:LinearAxis id="a1"
                minimum="30"
                maximum="50"/>
         </mx:verticalAxis>
         <mx:series>
            <mx:HLOCSeries
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="FRED">
            </mx:HLOCSeries>
         </mx:series>
      </mx:HLOCChart>
      <mx:Legend dataProvider="{mychart}"/>
  </s:Panel>
</s:Application>
```

Using ChartElement objects

The ChartElement class is the base class for anything that appears in the data area of the chart. All series objects (such as GridLines objects) are ChartElement objects. You can add ChartElement objects (such as images, grid lines, and strokes) to your charts by using the `backgroundElements` and `annotationElements` properties of the chart classes.

The `backgroundElements` property specifies an Array of ChartElement objects that appear beneath any data series rendered by the chart. The `annotationElements` property specifies an Array of ChartElement objects that appears above any data series rendered by the chart.

The ChartElement objects that you can add to a chart include supported image files, such as GIF, PNG, and JPEG.

The following example adds new grid lines as annotation elements to the chart and an image as the background element. When the user clicks the button, the annotation elements change:

```
<?xml version="1.0"?>
<!-- charts/AnnotationElements.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
      import mx.graphics.SolidColor;
```

```
     import mx.charts.GridLines;
     [Embed(source="../assets/butterfly.gif")]
     public var butterfly:Class;
     public function updateGridLines():void {
        var bgi:GridLines = new GridLines();
        var s:SolidColorStroke = new SolidColorStroke(0xff00ff, 3);
        bgi.setStyle("horizontalStroke",s);
        var c:SolidColor = new SolidColor(0x990033, .2);
        bgi.setStyle("horizontalFill",c);
        var c2:SolidColor = new SolidColor(0x999933, .2);
        bgi.setStyle("horizontalAlternateFill",c2);
        myChart.annotationElements = [bgi]
        var b:Object = new butterfly();
        b.alpha = .2;
        b.height = 150;
        b.width = 150;
        myChart.backgroundElements = [ b ];
     }
 ]]></fx:Script>
 <s:Panel title="Annotation Elements Example">
       <s:layout>
           <s:VerticalLayout/>
       </s:layout>
    <mx:ColumnChart id="myChart"
       dataProvider="{srv.lastResult.data.result}"
       showDataTips="true">
       <mx:horizontalAxis>
          <mx:CategoryAxis categoryField="month"/>
       </mx:horizontalAxis>
       <mx:series>
          <mx:ColumnSeries
               xField="month"
               yField="profit"
               displayName="Profit"/>
          <mx:ColumnSeries
               xField="month"
               yField="expenses"
               displayName="Expenses"/>
       </mx:series>
       <mx:annotationElements>
          <mx:GridLines>
           <mx:horizontalStroke>
               <mx:SolidColorStroke
```

```
                    color="#191970"
                    weight="2"
                    alpha=".3"/>
            </mx:horizontalStroke>
           </mx:GridLines>
        </mx:annotationElements>
        <mx:backgroundElements>
           <s:Image
                source="@Embed('../assets/butterfly.gif')"
                alpha=".2"/>
        </mx:backgroundElements>
     </mx:ColumnChart>
     <!--
     <mx:Legend dataProvider="{myChart}"/>
     -->
  </s:Panel>
  <s:Button id="b1"
    click="updateGridLines()"
    label="Update Grid Lines"/>
</s:Application>
```

In ActionScript, you can add an image to the chart and manipulate its properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BackgroundElementsWithActionScript.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();addButterfly()"
    height="600">
  <fx:Declarations>
     <!-- View source of the following page to see the structure of the data that Flex uses in
this example. -->
     <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
     <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
  </fx:Declarations>
  <fx:Script><![CDATA[
      import mx.collections.ArrayCollection;
      import mx.graphics.SolidColor;
      import mx.charts.GridLines;
      import mx.charts.ColumnChart;
      [Embed(source="../assets/butterfly.gif")]
      public var butterfly:Class;
      public function addButterfly():void {
         var b:Object = new butterfly();
         b.alpha = .2;
         myChart.backgroundElements = [ b ];
      }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Background Elements with ActionScript">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

For more information on working with grid lines, see "Using chart grid lines" on page 1218.

You can also add data graphics to your charts by adding the CartesianDataCanvas and PolarDataCanvas controls to your background or annotation elements Arrays. For more information, see "Drawing on chart controls" on page 1387.

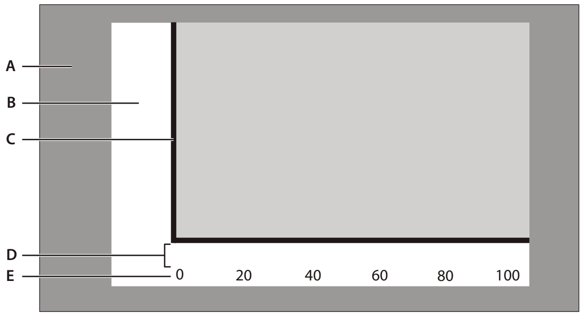
Setting padding properties

The padding properties of a chart define an area between the outside bounds of the chart control and its content. Flex draws only the background fill in the padding area.

You can set the padding values for a chart control by using the `paddingLeft` and `paddingRight` properties that the chart control inherits from the UIComponent class. You can also use the `paddingTop` and `paddingBottom` properties that the chart control inherits from the ChartBase class.

Flex charting control elements also have gutters. The *gutter* is the area between the padding area and the actual axis line. Flex draws labels, titles, and tick marks for the axes in the gutter of the chart. Chart controls adjust the gutters to accommodate these enhancements to the axis, but you can specify explicit gutter values.

The following example shows the locations of the gutters and the padding area on a chart:



A. *Padding area* ***B.*** *Gutter* ***C.*** *Axis* ***D.*** *Label gap* ***E.*** *Axis label*

The following style properties define the size of a chart’s gutters:

- `gutterLeft`
- `gutterRight`
- `gutterTop`
- `gutterBottom`

The default value of the gutter styles is `undefined`, which means that the chart determines appropriate values. Overriding the default value and explicitly setting gutter values can improve the speed of rendering charts, because Flex does not have to dynamically calculate the gutter size. However, it can also cause clipping of axis labels or other undesirable effects.

You set gutter styles on the chart control. The following example creates a region that is 50 pixels wide for the axis labels, titles, and tick marks, by explicitly setting the values of the `gutterLeft`, `gutterRight`, and `gutterBottom` style properties. It also sets the `paddingTop` property to 20.

```
<?xml version="1.0"?>
<!-- charts/GutterStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|ColumnChart {
        gutterLeft:50;
        gutterRight:50;
        gutterBottom:50;
        gutterTop:20;
     }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Setting Gutter Properties as Styles">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="column"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

Alternatively, you can set the gutter properties inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GutterProperties.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Setting Gutter Properties">
        <s:layout>
           <s:VerticalLayout/>
        </s:layout>
          <mx:ColumnChart id="column"
             dataProvider="{srv.lastResult.data.result}"
             showDataTips="true"
             gutterLeft="50"
             gutterRight="50"
             gutterBottom="50"
             gutterTop="20">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

In addition to the gutter and padding properties, you can set the `labelGap` property of a chart's axes. The `labelGap` property defines the distance, in pixels, between the tick marks and the axis labels. You set the `labelGap` property on the AxisRenderer tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LabelGaps.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Setting Label Gap Properties">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            gutterLeft="50"
            gutterRight="50"
            gutterBottom="50"
            gutterTop="20"
            showDataTips="true">
            <mx:horizontalAxis>
```

```
                <mx:CategoryAxis id="a1" categoryField="month"/>
            </mx:horizontalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer axis="{a1}" labelGap="20"/>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer axis="{a1}" labelGap="20"/>
            </mx:verticalAxisRenderers>
            <mx:series>
                <mx:ColumnSeries
                    xField="month"
                    yField="profit"
                    displayName="Profit"/>
                <mx:ColumnSeries
                    xField="month"
                    yField="expenses"
                    displayName="Expenses"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

You can specify the alignment of axis labels by using the `labelAlign` property on the AxisRenderer. For horizontal axis labels, you can specify `center`, `left`, and `right`. For example, in a ColumnChart control, if you set the value of the `labelAlign` property to `left`, Flex renders the labels at the `left` of the bottoms of the columns. For vertical axis labels, you can specify `center`, `top`, and `bottom`.

How different the locations of the labels are depends on the available space (for example, the width of the columns) and the font size of the labels.

The following example lets you change the label alignment for the vertical and horizontal axes.

```
<?xml version="1.0"?>
<!-- charts/ToggleLabelAlignment.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600" width="750">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script><![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
    private var aligns:ArrayCollection = new ArrayCollection(["center", "left", "right"]);
    [Bindable]
    private var aligns2:ArrayCollection = new ArrayCollection(["center", "top", "bottom"]);
    private function toggleLabelAlignment(s:String):void {
        if ( s == 'hor' ) {
            myHAR.setStyle("labelAlign", cb1.selectedItem);
        } else if ( s == 'ver' ) {
            myVAR.setStyle("labelAlign", cb2.selectedItem);
        }
    }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Toggling Label Alignment">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <s:VGroup>
         <mx:ColumnChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:horizontalAxis>
               <mx:CategoryAxis id="haxis" categoryField="month"/>
            </mx:horizontalAxis>
            <mx:verticalAxis>
                <mx:LinearAxis id="vaxis"/>
            </mx:verticalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer id="myHAR" axis="{haxis}"/>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer id="myVAR" axis="{vaxis}"/>
            </mx:verticalAxisRenderers>
            <mx:series>
               <mx:ColumnSeries xField="month" yField="profit" displayName="Profit"/>
               <mx:ColumnSeries xField="month" yField="expenses" displayName="Expenses"/>
            </mx:series>
         </mx:ColumnChart>
         <mx:Legend dataProvider="{myChart}"/>
     </s:VGroup>
     <s:Form>
        <s:FormItem label="Horizontal Labels">
            <s:ComboBox id="cb1" dataProvider="{aligns}"
                change="toggleLabelAlignment('hor')"/>
        </s:FormItem>
        <s:FormItem label="Vertical Labels">
            <s:ComboBox id="cb2" dataProvider="{aligns2}"
                change="toggleLabelAlignment('ver')"/>
        </s:FormItem>
     </s:Form>
  </s:Panel>
</s:Application>
```

Formatting tick marks

There are two types of tick marks on a Flex chart: major and minor. Major tick marks are the indications along an axis that correspond to an axis label. The text for the axis labels is often derived from the chart's data provider. Minor tick marks are those tick marks that appear between the major tick marks. Minor tick marks help the user to visualize the distance between the major tick marks.

You use the `tickPlacement` and `minorTickPlacement` properties of the AxisRenderer object to determine whether or not Flex displays tick marks and where Flex displays tick marks.

The following table describes valid values of the `tickPlacement` and `minorTickPlacement` properties:

Value	Description
`cross`	Places tick marks across the axis.
`inside`	Places tick marks on the inside of the axis line.
`none`	Hides tick marks.
`outside`	Places tick marks on the outside of the axis line.

Flex also lets you set the length of tick marks and the number of minor tick marks that appear along the axis. The following table describes the properties that define the length of tick marks on the chart's axes:

Property	Description
`tickLength`	The length, in pixels, of the major tick mark from the axis.
`minorTickLength`	The length, in pixels, of the minor tick mark from the axis.

The minor tick marks overlap the placement of major tick marks. So, if you hide major tick marks but still show minor tick marks, the minor tick marks appear at the regular tick-mark intervals.

The following example sets tick marks to the inside of the axis line, sets the tick mark's length to 12 pixels, and hides minor tick marks:

```
<?xml version="1.0"?>
<!-- charts/TickStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600" width="700">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
  <fx:Style>
     .myAxisStyle {
        placement:bottom;
        minorTickPlacement:none;
        tickLength:12;
        tickPlacement:inside;
        tickStroke: myStroke;
     }
  </fx:Style>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Tick Styles">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
    <mx:HLOCChart id="mychart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
    <mx:horizontalAxisRenderers>
        <mx:AxisRenderer axis="{a2}" styleName="myAxisStyle">
               <mx:tickStroke>
                    <mx:SolidColorStroke color="#999966" weight="1"/>
                </mx:tickStroke>
        </mx:AxisRenderer>
    </mx:horizontalAxisRenderers>
    <mx:verticalAxis>
        <mx:LinearAxis id="a1" minimum="30" maximum="50"/>
    </mx:verticalAxis>
    <mx:horizontalAxis>
        <mx:LinearAxis id="a2"/>
    </mx:horizontalAxis>
    <mx:series>
        <mx:HLOCSeries openField="open" highField="high"
            lowField="low" closeField="close"
            displayName="Ticker Symbol: FRED"/>
     </mx:series>
  </mx:HLOCChart>
  <mx:Legend dataProvider="{mychart}"/>
  </s:Panel>
</s:Application>
```

Formatting axis lines

Axes have lines to which the tick marks are attached. You can use style properties to hide these lines or change the width of the lines.

To hide the axis line, set the value of the `showLine` property on the AxisRenderer object to `false`. The default value is `true`. The following example sets `showLine` to `false`:

```
<?xml version="1.0"?>
<!-- charts/DisableAxisLines.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
          <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
          <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Line Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:LineChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis id="a1" categoryField="month"/>
            </mx:horizontalAxis>
               <mx:verticalAxis>
                    <mx:LinearAxis id="a2"/>
               </mx:verticalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    showLine="false"/>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer axis="{a2}" showLine="false"/>
            </mx:verticalAxisRenderers>
            <mx:series>
                <mx:LineSeries yField="profit" displayName="Profit"/>
                <mx:LineSeries yField="expenses" displayName="Expenses"/>
            </mx:series>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

You can also apply the `showLine` property as a CSS style property.

You can change the `width`, `color`, and `alpha` of the axis line with the `<mx:axisStroke>` tag. You use an `<mx:SolidColorStroke>` child tag to define these properties or define a stroke and then bind it to the `axisStroke` object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/StyleAxisLines.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <mx:SolidColorStroke id="axisStroke" color="#884422"
            weight="8" alpha=".75" caps="square"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Line Chart">
         <s:layout>
             <s:HorizontalLayout/>
         </s:layout>
        <mx:LineChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis id="a1" categoryField="month"/>
            </mx:horizontalAxis>
            <mx:verticalAxis>
                <mx:LinearAxis id="a2"/>
            </mx:verticalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer axis="{a1}">
                    <mx:axisStroke>{axisStroke}</mx:axisStroke>
                </mx:AxisRenderer>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer axis="{a2}">
                    <mx:axisStroke>
                        <mx:SolidColorStroke color="#884422"
                            weight="8" alpha=".75" caps="square"/>
                    </mx:axisStroke>
                </mx:AxisRenderer>
            </mx:verticalAxisRenderers>
            <mx:series>
                <mx:LineSeries yField="profit" displayName="Profit"/>
                <mx:LineSeries yField="expenses" displayName="Expenses"/>
            </mx:series>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

For more information about strokes, see “Using strokes with chart controls” on page 1192.

You can change the placement of the axis lines (for example, move the axis line from the bottom of the chart control to the top) by using the `placement` property of the AxisRenderer. For more information, see “Positioning chart axes” on page 1226.

You can also apply filters to axis lines to further customize their appearance. For more information, see “Using filters with chart controls” on page 1213.

Using strokes with chart controls

You use the SolidColorStroke class with the chart series and grid lines to control the properties of the lines that Flex uses to draw chart elements.

You can define strokes in MXML inside the `<fx:Declarations>` block so that you can reuse the stroke. You can also define an individual stroke inside an MXML tag.

The following table describes the properties that you use to control the appearance of strokes:

Property	Description
`color`	Specifies the color of the line as a hexadecimal value. The default value is `0x000000`, which corresponds to black.
`weight`	Specifies the width of the line, in pixels. The default value is 0, which corresponds to a hairline.
`alpha`	Specifies the transparency of a line. Valid values are 0 (invisible) through 100 (opaque). The default value is 100.

The following example defines a line width of 2 pixels, one with a dark gray border (0x808080) and the other with a light gray border (0xC0C0C0) for the borders of chart items in a BarChart control’s BarSeries:

```
<?xml version="1.0"?>
<!-- charts/BasicBarStroke.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="Profit">
            <mx:stroke>
                <mx:SolidColorStroke
                    color="0x808080"
                    weight="2" alpha=".8"/>
            </mx:stroke>
           </mx:BarSeries>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="Expenses">
            <mx:stroke>
                <mx:SolidColorStroke
                    color="0xC0C0C0"
                    weight="2" alpha=".8"/>
            </mx:stroke>
           </mx:BarSeries>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Defining AxisRenderer properties with strokes

You can use strokes to define tick marks and other properties of an AxisRenderer, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/AxisRendererStrokes.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
  <fx:Declarations>
      <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
      <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
      <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
  </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="HighLowOpenClose Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    <mx:HLOCChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer
                placement="bottom"
                canDropLabels="true"
                tickPlacement="inside"
                tickLength="10"
                minorTickPlacement="inside"
                minorTickLength="5"
                axis="{a1}">
                <mx:axisStroke>
                    <mx:SolidColorStroke color="#000080" weight="1"/>
                </mx:axisStroke>
                <mx:tickStroke>
                    <mx:SolidColorStroke color="#000060" weight="1"/>
                </mx:tickStroke>
                <mx:minorTickStroke>
                    <mx:SolidColorStroke color="#100040" weight="1"/>
                </mx:minorTickStroke>
            </mx:AxisRenderer>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxis>
            <mx:LinearAxis id="a1" minimum="30"  maximum="50"/>
        </mx:verticalAxis>
        <mx:series>
            <mx:HLOCSeries
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="FRED">
            </mx:HLOCSeries>
        </mx:series>
    </mx:HLOCChart>
    <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Notice that any time you use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer's `axis` property.

You can define a SolidColorStroke object using an MXML tag, and then bind that SolidColorStroke to the chart's renderer properties. For an example, see "Applying styles by binding tag definitions" on page 1177.

Using strokes in ActionScript

You can instantiate and manipulate a SolidColorStroke object in ActionScript by using the mx.graphics.SolidColorStroke class. You can then use the setStyle() method to apply the SolidColorStroke object to the chart, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ActionScriptStroke.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns="*"
    creationComplete="srv.send()"
    height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
     import mx.graphics.SolidColorStroke;
     public function changeStroke(e:Event):void {
        var s:SolidColorStroke = new SolidColorStroke(0x001100,2);
        s.alpha = .5;
        s.color = 0x0000FF;
        har1.setStyle("axisStroke",s);
        var1.setStyle("axisStroke",s);
     }
  ]]></fx:Script>
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <mx:SolidColorStroke id="baseAxisStroke"
            color="0x884422"
            weight="10"
            alpha=".25"
            caps="square"/>
    </fx:Declarations>
  <s:Panel title="Column Chart">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}" 
showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis id="a1" categoryField="month"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
          <mx:LinearAxis id="a2"/>
        </mx:verticalAxis>
        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer id="har1" axis="{a1}">
                <mx:axisStroke>{baseAxisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer id="var1" axis="{a2}">
```

```
                <mx:axisStroke>{baseAxisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:verticalAxisRenderers>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
  <s:Button id="b1"
    click="changeStroke(event)"
    label="Change SolidColorStroke"/>
</s:Application>
```

Defining strokes for LineSeries and AreaSeries

Some chart series have more than one stroke-related style property. For LineSeries, you use the `stroke` style property to define a style for the chart item's renderer. You use the `lineStroke` property to define the stroke of the actual line segments.

The following example creates a thick blue line for the LineChart control's line segments, with large red boxes at each data point, which use the CrossItemRenderer object as their renderer:

```
<?xml version="1.0"?>
<!-- charts/LineSeriesStrokes.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
            yField="profit"
            displayName="Profit">
            <mx:itemRenderer>
                <fx:Component>
                    <mx:CrossItemRenderer scaleX="1.5" scaleY="1.5"/>
                </fx:Component>
            </mx:itemRenderer>
            <mx:fill>
                <mx:SolidColor color="0x0000FF"/>
            </mx:fill>
            <mx:stroke>
                <mx:SolidColorStroke color="0xFF0066" alpha="1"/>
            </mx:stroke>
            <mx:lineStroke>
                <mx:SolidColorStroke color="0x33FFFF" weight="5" alpha=".8"/>
            </mx:lineStroke>
           </mx:LineSeries>
           <mx:LineSeries yField="expenses" displayName="Expenses"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Similarly, with the AreaSeries class, you use the `stroke` property to set a style for the chart item's renderer. You use the `areaStroke` style property to define the stroke of the line that defines the area.

Using fills with chart controls

When charting multiple data series, or just to improve the appearance of your charts, you can control the fill for each series in the chart or each item in a series. The fill lets you specify a pattern that defines how Flex draws the chart element. You also use fills to specify the background colors of the chart or bands of background colors defined by the grid lines. Fills can be solid or can use linear and radial gradients. A *gradient* specifies a gradual color transition in the fill color.

You use fills in the following ways:

- `fill` Set the value of the `fill` property on a series to a single fill. This applies that fill to all chart items in that series. For example, all columns in a single series will have the same fill.
- `fills` Set the value of the `fills` property on a series to an Array of fills. This applies a different fill from the Array to each chart item in the series. For example, each column in a series will have a different fill. The LineSeries and LineRenderer objects are not affected by the `fill` property's settings.
- `fillFunction` Set the value of the `fillFunction` property on a series to point to a custom method. This method returns a fill based on the values of the chart item in the series. The return value of this function takes precedence over fills specified as styles. For information on using the `fillFunction` property, see "Using per-item fills" on page 1299.

All series except the HLOCSeries class support setting the fill-related properties.

If you use the `fills` property or the `fillFunction` to define the fills of chart items, and you want a legend, you must manually create the Legend object for that chart. For more information on creating Legend objects, see “Using Legend controls” on page 1316.

One of the most common uses of a fill is to control the color of the chart when you have multiple data series in a chart. The following example uses the `fill` property to set the color for each ColumnSeries object in a ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/ColumnFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit">
               <mx:fill>
                    <mx:SolidColor color="0x336699"/>
               </mx:fill>
           </mx:ColumnSeries>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses">
            <mx:fill>
                <mx:SolidColor color="0xFF99FF"/>
            </mx:fill>
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

If you do not explicitly define different fills for multiple data series, Flex chooses solid colors for you.

You can also specify an array of colors using the `fills` property of the series. This property takes an Array of objects that define the fills. You can use this property to define a different color for each item in the series. If you do not define as many colors in the Array as there are items in the series, the chart starts with the first item in the Array on the next item.

The following example defines two Arrays of SolidColor objects. The first Array defines the colors of the items in the first series in the chart and the second Array defines the colors of the items in the second series. All of the fills in this example are partially transparent (the alpha value in the SolidColor constructor is .5).

```
<?xml version="1.0"?>
<!-- charts/SimpleFillsExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script>
    <![CDATA[
     import mx.graphics.SolidColor;

     // Make all fills partially transparent by
     // setting the alpha to .5.
     [Bindable]
     private var warmColorsArray:Array = new Array(
        new SolidColor(0xFF0033, .5),
        new SolidColor(0xFF0066, .5),
        new SolidColor(0xFF0099, .5)
     );
     [Bindable]
     private var coolColorsArray:Array = new Array(
        new SolidColor(0x3333CC, .5),
        new SolidColor(0x3366CC, .5),
        new SolidColor(0x3399CC, .5)
     );

    ]]>
  </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"
                fills="{warmColorsArray}">
           </mx:ColumnSeries>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"
                fills="{coolColorsArray}">
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

This example does not have a legend. If you use the `fills` property or the `fillFunction` to define the fills of chart items, and you want a legend, you must manually create the Legend object for that chart. For more information on creating Legend objects, see “Using Legend controls” on page 1316.

With the PieSeries, you typically use a single Array of fills to specify how Flex should draw the individual wedges. For example, you can give each wedge that represents a PieSeries its own color, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/PieFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Pie Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PieChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <mx:PieSeries
                field="amount"
                nameField="item"
                labelPosition="callout">
                <mx:fills>
                    <mx:SolidColor color="0xCC66FF" alpha=".8"/>
                    <mx:SolidColor color="0x9966CC" alpha=".8"/>
                    <mx:SolidColor color="0x9999CC" alpha=".8"/>
                    <mx:SolidColor color="0x6699CC" alpha=".8"/>
                    <mx:SolidColor color="0x669999" alpha=".8"/>
                    <mx:SolidColor color="0x99CC99" alpha=".8"/>
                </mx:fills>
            </mx:PieSeries>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

This example also shows how you use the `<mx:fills>` tag inside a series to define an Array of fills for that series.

You can also use fills to set the background of the charts. You do this by adding an `<mx:fill>` child tag to the chart tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BackgroundFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
  <fx:Declarations>
     <!-- View source of the following page to see the structure of the data that Flex uses in
this example. -->
     <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
     <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
  </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Background Fill">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:fill>
           <mx:SolidColor color="0x66CCFF" alpha=".5"/>
        </mx:fill>
        <mx:series>
           <mx:BarSeries xField="expenses" displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Setting fills with CSS

You can use the `fill` or `fills` style properties in an `<fx:Style>` declaration by using CSS syntax. You can use a type or a class selector. The following example sets the fill of the custom `myBarChartStyle` class selector to `#FF0000`:

```
<?xml version="1.0"?>
<!-- charts/BackgroundFillsCSS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
  <fx:Declarations>
     <!-- View source of the following page to see the structure of the data that Flex uses in
this example. -->
     <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
     <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
  </fx:Declarations>
  <fx:Style>
     .myBarChartStyle {
        fill:#FF0000;
     }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Background Fill">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        styleName="myBarChartStyle">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries xField="expenses" displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Flex converts the fill style in the class selector to a SolidColor object.

The following example defines colors for each of the series by setting the value of the `fill` style property in the custom class selectors:

```
<?xml version="1.0"?>
<!-- charts/SimpleCSSFillsExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Style>
        .myRedColumnSeries {
            fill:#FF0033;
        }
        .myGreenColumnSeries {
            fill:#33FF00;
        }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries 
                xField="month" 
                yField="profit" 
                displayName="Profit"
                styleName="myRedColumnSeries">
           </mx:ColumnSeries>
           <mx:ColumnSeries 
                xField="month" 
                yField="expenses" 
                displayName="Expenses"
                styleName="myGreenColumnSeries">
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To specify an Array for the `fills` property in CSS, you use a comma-separated list, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CSSFillsArrayExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Style>
        .myRedColumnSeries {
            fills: #FF0033, #FF3333, #FF6633;
        }
        .myGreenColumnSeries {
            fills: #33FF00, #33FF33, #33FF66;
        }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"
                styleName="myRedColumnSeries">
           </mx:ColumnSeries>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"
                styleName="myGreenColumnSeries">
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Using gradient fills with chart controls

Flex provides several classes that let you specify gradient fills. You use either the LinearGradient class or the RadialGradient class, along with the GradientEntry class to specify a gradient fill. The following table describes these classes:

Class	Description
LinearGradient	Defines a gradient fill that starts at a boundary of the chart element. You specify an array of GradientEntry objects to control gradient transitions with the LinearGradient object.
RadialGradient	Defines a gradient fill that radiates from the center of a chart element. You specify an array of GradientEntry objects to control gradient transitions with the RadialGradient object.
GradientEntry	Defines the objects that control the gradient transition. Each GradientEntry object contains the following properties: • `color` Specifies a color value. • `alpha` Specifies the transparency. Valid values are 0 (invisible) through 1 (opaque). The default value is 1. • `ratio` Specifies where in the chart, as a percentage, Flex starts the transition to the next `color`. For example, if you set the `ratio` property to .33, Flex begins the transition 33% of the way through the chart. If you do not set the `ratio` property, Flex tries to evenly apply values based on the `ratio` properties for the other GradientEntry objects. Valid values range from 0 to 1.

The following example uses a LinearGradient class with gradient fills for the chart's background:

```
<?xml version="1.0"?>
<!-- charts/GradientFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Gradient Background Fill">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:fill>
           <mx:LinearGradient>
```

```
            <mx:entries>
                <mx:GradientEntry
                    color="0xC5C551"
                    ratio="0"
                    alpha="1"/>
                <mx:GradientEntry
                    color="0xFEFE24"
                    ratio=".33"
                    alpha="1"/>
                <mx:GradientEntry
                    color="0xECEC21"
                    ratio=".66"
                    alpha="1"/>
            </mx:entries>
           </mx:LinearGradient>
        </mx:fill>
        <mx:series>
           <mx:BarSeries xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The LinearGradient object takes a single attribute, `angle`. By default, it defines a transition from left to right across the chart. Use the `angle` property to control the direction of the transition. For example, a value of 180 causes the transition to occur from right to left, rather than from left to right.

To add gradient fills to a chart in ActionScript, you define GradientEntry objects, assign values to their properties (such as `ratio` and `color`), and add them as entries to a gradient's entries Array. You then apply the gradient to the chart's fill by using the `setStyle()` method. The following example shows how to do this in ActionScript:

```
<?xml version="1.0"?>
<!-- charts/GradientFillsInActionScript.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.graphics.LinearGradient;
     import mx.graphics.GradientEntry;
     import mx.collections.ArrayCollection;

     private function styleChart():void {
          var ge1:GradientEntry = new GradientEntry(0xCCFF99, 0);
          var ge2:GradientEntry = new GradientEntry(0x99FF00, .33);
          var ge3:GradientEntry = new GradientEntry(0x669900, .66);

          var lg1:LinearGradient = new LinearGradient();
          lg1.entries = [ge1, ge2, ge3];

          myChart.setStyle("fill", lg1);
     }
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Gradient Entries in ActionScript Fill">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        creationComplete="styleChart()">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
  </s:Panel>
</s:Application>
```

You can set other properties on gradients, such as `angle`, to change the appearance of chart fills. The following example sets the `angle` property to 90, which specifies that the transition occurs from the top of the chart to the bottom.

```
<?xml version="1.0"?>
<!-- charts/GradientFillsAngled.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Angled Gradient Background Fill">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
          dataProvider="{srv.lastResult.data.result}"
          showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:fill>
           <mx:LinearGradient rotation="90">
            <mx:entries>
                <mx:GradientEntry
                    color="0xC5C551"
                    ratio="0"
                    alpha="1"/>
                <mx:GradientEntry
                    color="0xFEFE24"
                    ratio=".33"
                    alpha="1"/>
                <mx:GradientEntry
                    color="0xECEC21"
                    ratio=".66"
                    alpha="1"/>
            </mx:entries>
           </mx:LinearGradient>
        </mx:fill>
        <mx:series>
           <mx:BarSeries xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Using different alpha values with fills

When charting multiple data series, you can define the series to overlap. For example, the column chart lets you display the columns next to each other or overlap them for multiple data series. To overlap series, you set the `type` property of the chart to `overlaid`. The same is true for an area series.

When you have multiple data series that overlap, you can specify that the fill for each series has an alpha value less than 100%, so that the series have a level of transparency. The valid values for the `alpha` property are 0 (invisible) through 1 (opaque).

You cannot specify alpha values for fills if you apply the fills using CSS.

The following example defines an area chart in which each series in the chart uses a solid fill with the same level of transparency:

```
<?xml version="1.0"?>
<!-- charts/AlphaFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Area Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:AreaChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="profit"
                displayName="Profit">
                <mx:areaStroke>
                    <mx:SolidColorStroke
                        color="0x9A9A00"
                        weight="2"/>
                </mx:areaStroke>
                <mx:areaFill>
                    <mx:SolidColor
                        color="0x7EAEFF"
                        alpha=".3"/>
                </mx:areaFill>
           </mx:AreaSeries>
           <mx:AreaSeries
                yField="expenses"
                displayName="Expenses">
                <mx:areaStroke>
                    <mx:SolidColorStroke
                        color="0x9A9A00"
                        weight="2"/>
                </mx:areaStroke>
                <mx:areaFill>
                    <mx:SolidColor
                        color="0xAA0000"
                        alpha=".3"/>
                </mx:areaFill>
           </mx:AreaSeries>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

If you define a gradient fill, you can set the `alpha` property on each entry in the Array of `<mx:GradientEntry>` tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GradientAlphaFills.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Area Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:AreaChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
            <mx:AreaSeries yField="profit" displayName="Profit">
                <mx:areaStroke>
                    <mx:SolidColorStroke color="0x9A9A00" weight="2"/>
                </mx:areaStroke>
                <mx:areaFill>
                    <mx:LinearGradient rotation="90">
                        <mx:entries>
                            <mx:GradientEntry
                                color="0xC5C551"
                                ratio="0"
                                alpha="1"/>
                            <mx:GradientEntry
                                color="0xFEFE24"
                                ratio=".33"
                                alpha="1"/>
                            <mx:GradientEntry
                                color="0xECEC21"
                                ratio=".66"
                                alpha=".2"/>
                        </mx:entries>
                    </mx:LinearGradient>
                </mx:areaFill>
           </mx:AreaSeries>
           <mx:AreaSeries
                yField="expenses"
                displayName="Expenses">
                <mx:areaStroke>
                    <mx:SolidColorStroke color="0x9A9A00" weight="2"/>
                </mx:areaStroke>
                <mx:areaFill>
                    <mx:LinearGradient rotation="90">
                        <mx:entries>
                            <mx:GradientEntry
                                color="0xAA0000"
                                ratio="0"
                                alpha="1"/>
                            <mx:GradientEntry
                                color="0xCC0000"
                                ratio=".33"
                                alpha="1"/>
                            <mx:GradientEntry
                                color="0xFF0000"
                                ratio=".66"
                                alpha=".2"/>
                        </mx:entries>
                    </mx:LinearGradient>
                </mx:areaFill>
           </mx:AreaSeries>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

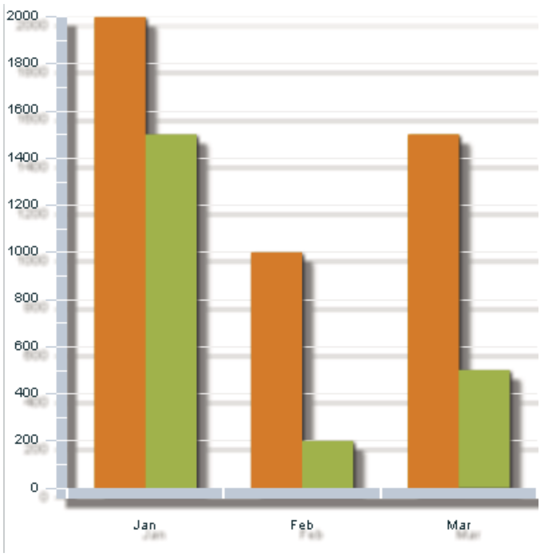
In this example, you make the last gradient color in the series partially transparent by setting its `alpha` property to .2.

Using filters with chart controls

You can add filters such as drop shadows to your chart elements by using the classes in the spark.filters.* package. These filters include:

- BevelFilter
- BitmapFilter
- BlurFilter
- ColorMatrixFilter
- ConvolutionFilter
- DisplacementMapFilter
- DropShadowFilter
- GlowFilter
- GradientBevelFilter
- GradientGlowFilter
- ShaderFilter

You can apply filters to the chart control itself, or to each chart series. When you apply a filter to a chart control, the filter is applied to all aspects of that chart control, including gridlines, axis labels, and each data point in the series. The following image shows a drop shadow filter applied to a ColumnChart control:



You wrap the filter class in an `<mx:filters>` or `<s:filters>` tag to define the array of filters applied to that control. If the control is an MX control, you use the `<mx:filters>` tag. If the control is a Spark control, you use the `<s:filters>` tag. In the case of charting components, you use the `<mx:filters>` tag because the charts are in the MX component set. At compile time, the MX filters are mapped to the Spark filters.

The following example applies a custom drop shadow filter to a ColumnChart control. The result is that every element, including the grid lines, axis labels, and columns, has a background shadow.

```
<?xml version="1.0"?>
<!-- charts/ColumnWithDropShadow.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="ColumnChart with drop shadow example">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="column" dataProvider="{srv.lastResult.data.result}" 
showDataTips="true">
        <!-- Add a custom drop shadow filter to the ColumnChart control. -->
        <mx:filters>
           <s:DropShadowFilter 
                distance="10" 
                color="0x666666" 
                alpha=".8"/>
        </mx:filters>
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries 
                xField="month" 
                yField="profit" 
                displayName="Profit"/>
           <mx:ColumnSeries 
                xField="month" 
                yField="expenses" 
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

For more information on filters, see “Using filters” on page 1556.

Adding a drop shadow filter to some chart controls can have unexpected consequences. For example, if you add a drop shadow filter to a PieChart control, Flex renders that drop shadow filter in addition to the default drop shadow filter on the PieSeries.

You can remove filters by setting the filters Array to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ClearFilters.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
      import mx.collections.ArrayCollection;
      [Bindable]
      public var expenses:ArrayCollection = new ArrayCollection([
         {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
         {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
         {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
      ]);
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel id="p1" title="Line chart with no filters">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
      <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true">

         <!-- Clear the filters array. -->
         <mx:seriesFilters>
            <fx:Array/>
         </mx:seriesFilters>
         <mx:horizontalAxis>
            <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
         </mx:horizontalAxis>
         <mx:series>
            <mx:LineSeries
                yField="Profit"
                displayName="Profit"/>
            <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"/>
            <mx:LineSeries
                yField="Amount"
                displayName="Amount"/>
         </mx:series>
      </mx:LineChart>
  </s:Panel>
</s:Application>
```

The following example creates a PieChart control and applies a drop shadow to it; it also removes the default drop shadow filter from the PieSeries so that there is a single drop shadow:

```
<?xml version="1.0"?>
<!-- charts/PieChartShadow.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Pie Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PieChart id="pie" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <!-- Add a custom drop shadow to the PieChart control -->
        <mx:filters>
           <s:DropShadowFilter 
                distance="10" 
                color="0x666666" 
                alpha=".8"/>
        </mx:filters>
        <mx:series>
           <fx:Array>
                <mx:PieSeries field="amount" nameField="item" 
                labelPosition="callout" explodeRadius=".2">
                    <!-- Clear default shadow on the PieSeries -->
                    <mx:filters>
                        <fx:Array/>
                    </mx:filters>
                </mx:PieSeries>
           </fx:Array>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </s:Panel>
</s:Application>
```

You can add filters to individual chart elements that are display objects, such as series, grid lines, legend items, and axes. The following example defines set of filters, and then applies them to various chart elements:

```
<?xml version="1.0"?>
<!-- charts/MultipleFilters.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();createFilters();"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import spark.filters.*;
     import flash.filters.BitmapFilterType;
     import flash.filters.BitmapFilterQuality;

     private var myBlurFilter:BlurFilter;
     private var myGlowFilter:GlowFilter;
     private var myBevelFilter:BevelFilter;
     private var myDropShadowFilter:DropShadowFilter;
     private var color:Number = 0xFF33FF;
     public function applyFilters():void {
        // Apply filters to series, grid lines, legend, and axis.
        myGridlines.filters = [myBlurFilter];
        myLegend.filters = [myGlowFilter];
        myAxisRenderer.filters = [myBevelFilter];
        s1.filters = [myDropShadowFilter];
        s2.filters = [myDropShadowFilter];
     }
     public function createFilters():void {
        // Define filters.
        myBlurFilter = new BlurFilter(4,4,1);
        myGlowFilter = new GlowFilter(color, .8, 6, 6,
            2, 1, false, false);
        myDropShadowFilter = new DropShadowFilter(15, 45,
            color, 0.8, 8, 8, 0.65, 1, false, false);
        myBevelFilter = new BevelFilter(5, 45, color, 0.8,
            0x333333, 0.8, 5, 5, 1, BitmapFilterQuality.HIGH,
            BitmapFilterType.INNER, false);
        applyFilters();
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Applying Multiple Filters">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:backgroundElements>
            <mx:GridLines id="myGridlines"
                horizontalChangeCount="1"
                verticalChangeCount="1"
                gridDirection="both"/>
        </mx:backgroundElements>

        <mx:horizontalAxis>
            <mx:CategoryAxis id="a1" categoryField="month"/>
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer
                id="myAxisRenderer"
                placement="bottom"
                canDropLabels="true"
                axis="{a1}">
                <mx:axisStroke>
                    <mx:SolidColorStroke color="#000080" weight="10"/>
                </mx:axisStroke>
                <mx:tickStroke>
                    <mx:SolidColorStroke color="#000060" weight="5"/>
                </mx:tickStroke>
                <mx:minorTickStroke>
                    <mx:SolidColorStroke color="#100040" weight="5"/>
                </mx:minorTickStroke>
            </mx:AxisRenderer>
        </mx:horizontalAxisRenderers>
        <mx:series>
           <mx:ColumnSeries id="s1"
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries id="s2"
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend id="myLegend" dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Using chart grid lines

All charts except the PieChart control have grid lines by default. You can control those grid lines with the CSS `gridLinesStyleName` property, and with the chart series' `backgroundElements` and `annotationElements` properties.

You can include horizontal, vertical, or both grid lines in your chart with the GridLines object. You can set these behind the data series by using the chart's `backgroundElements` property or in front of the data series by using the `annotationElements` property.

The following example turns on grid lines in both directions and applies them to the chart:

```
<?xml version="1.0"?>
<!-- charts/GridLinesBoth.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <fx:Array id="bge">
            <mx:GridLines gridDirection="both"/>
        </fx:Array>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}"
        backgroundElements="{bge}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Note: *The* `annotationElements` *property refers to any chart elements that appear in the foreground of your chart, and the* `backgroundElements` *property refers to any chart elements that appear behind the chart's data series.*

You can also define the grid lines inside each chart control's definition, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GridLinesBothInternal.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
        <mx:backgroundElements>
           <mx:GridLines gridDirection="both"/>
        </mx:backgroundElements>
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To define the fills and strokes for grid lines, you use the `horizontalStroke`, `verticalStroke`, `horizontalFill`, and `verticalFill` properties. The following properties also define the appearance of grid lines:

- `horizontalAlternateFill`
- `horizontalChangeCount`
- `horizontalOriginCount`
- `horizontalShowOrigin`

- `horiztontalTickAligned`
- `verticalAlternateFill`
- `verticalChangeCount`
- `verticalOriginCount`
- `verticalShowOrigin`
- `verticalTickAligned`

For information on working with strokes, see “Using strokes with chart controls” on page 1192. For more information on using the `backgroundElements` and `annotationElements` properties, see “Using ChartElement objects” on page 1179.

You can manipulate the appearance of the grid lines directly in MXML, with ActionScript, or with CSS. The following sections describe techniques for formatting grid lines for chart objects.

Formatting chart grid lines with MXML

To control the appearance of the grid lines, you can specify an array of GridLines objects as MXML tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <fx:Array id="bge">
            <mx:GridLines
                horizontalChangeCount="1"
                verticalChangeCount="1"
                gridDirection="both"
            >
                <mx:horizontalStroke>
                    <mx:SolidColorStroke weight="3"/>
                </mx:horizontalStroke>
                <mx:verticalStroke>
                    <mx:SolidColorStroke weight="3"/>
                </mx:verticalStroke>
                <mx:horizontalFill>
                    <mx:SolidColor color="0x99033" alpha=".66"/>
                </mx:horizontalFill>
            </mx:GridLines>
        </fx:Array>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
```

```
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}"
        backgroundElements="{bge}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

This example uses the `changeCount` property to specify that Flex draws grid lines at every tick mark along the axis, and sets the `gridDirection` property to `both`. This causes Flex to draw grid lines both horizontally and vertically. You could also specify `horizontal` or `vertical` as values for the `gridDirection` property.

To remove grid lines entirely, you can set the `backgroundElements` property to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/NoGridLines.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <fx:Array id="bge">
        </fx:Array>
    </fx:Declarations>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="BubbleChart control with no grid lines">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
    <mx:BubbleChart id="bc"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        backgroundElements="{bge}">
        <mx:series>
            <mx:BubbleSeries
                xField="profit"
                yField="expenses"
                radiusField="amount"
                displayName="Profit"/>
        </mx:series>
    </mx:BubbleChart>
    <mx:Legend dataProvider="{bc}"/>
  </s:Panel>
</s:Application>
```

You can also change the appearance of grid lines by using filters such as a drop shadow, glow, or bevel. For more information, see "Using filters with chart controls" on page 1213.

Formatting chart grid lines with CSS

You can set the style of grid lines by applying a CSS style to the GridLines object. The following example applies the `myStyle` style to the grid lines:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatCSS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->

        <!-- Background element array. -->
        <fx:Array id="bge">
            <mx:GridLines styleName="myStyle">
                <mx:horizontalStroke>
                    <mx:SolidColorStroke weight="3"/>
                </mx:horizontalStroke>
                <mx:verticalStroke>
                    <mx:SolidColorStroke weight="3"/>
                </mx:verticalStroke>
            </mx:GridLines>
        </fx:Array>
    </fx:Declarations>
```

```
  <fx:Style>
     .myStyle {
        gridDirection:"both";
        horizontalShowOrigin:true;
        horizontalTickAligned:false;
        horizontalChangeCount:1;
        verticalShowOrigin:false;
        verticalTickAligned:true;
        verticalChangeCount:1;
        horizontalFill:#990033;
        horizontalAlternateFill:#00CCFF;
     }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}"
        backgroundElements="{bge}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Formatting chart grid lines with ActionScript

You can manipulate the GridLines at run time with ActionScript. The following example adds filled grid lines in front of and behind the chart's series:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatActionScript.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.graphics.SolidColor;
     import mx.graphics.SolidColorStroke;
     import mx.charts.GridLines;
     [Bindable]
     public var bge:GridLines;
     public function addGridLines():void {
        bge = new GridLines();
        var s:SolidColorStroke = new SolidColorStroke(0xff00ff, 2);
        bge.setStyle("horizontalStroke", s);
        var f:SolidColor = new SolidColor(0x990033, .3);
        bge.setStyle("horizontalFill",f);
        var f2:SolidColor = new SolidColor(0x336699, .3);
        bge.setStyle("horizontalAlternateFill",f2);
        myChart.backgroundElements = [bge];
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
```

```
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}"
        creationComplete="addGridLines()">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Positioning chart axes

You can place axes on the left, right, top, or bottom of the chart control. This includes the axis line as well as labels and any tick marks you added. You can place horizontal axes at the top or bottom of the chart (or both, if you have multiple vertical axes). You can place vertical axes at the left or right of the chart (or both, if you have multiple horizontal axes).

To change the location of axes, you use the `placement` property of the AxisRenderer. Valid values for this property for a vertical axis are `top` and `bottom`. Valid values for this property for a horizontal axis are `left` and `right`.

The following example creates a ColumnChart control with the default axis locations (bottom and left). You can select new locations by using the ComboBox controls at the bottom of the panel.

```
<?xml version="1.0"?>
<!-- charts/AxisPlacementExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="700">
  <fx:Declarations>
     <!-- View source of the following page to see the structure of the data that Flex uses in
this example. -->
     <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
     <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
  </fx:Declarations>
  <fx:Script><![CDATA[
      import mx.collections.ArrayCollection;

     [Bindable]
      private var horChoices:ArrayCollection = new ArrayCollection(["bottom","top"]);
     [Bindable]
      private var vertChoices:ArrayCollection = new ArrayCollection(["left","right"]);
  ]]></fx:Script>
```

```
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Variable Axis Placement">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer id="horAxisRend"
                axis="{axis1}"
                placement="{myHorBox.selectedItem}"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
            <mx:AxisRenderer id="vertAxisRend"
                axis="{axis2}"
                placement="{myVertBox.selectedItem}"/>
        </mx:verticalAxisRenderers>

        <mx:horizontalAxis>
           <mx:CategoryAxis id="axis1"
                categoryField="month"/>
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis id="axis2"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
                xField="month" yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month" yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

    <s:Form>
        <s:FormItem label="Horizontal Axis Location:">
            <s:ComboBox id="myHorBox" dataProvider="{horChoices}"/>
        </s:FormItem>
        <s:FormItem label="Vertical Axis Location:">
            <s:ComboBox id="myVertBox" dataProvider="{vertChoices}"/>
        </s:FormItem>
    </s:Form>

  </s:Panel>
</s:Application>
```

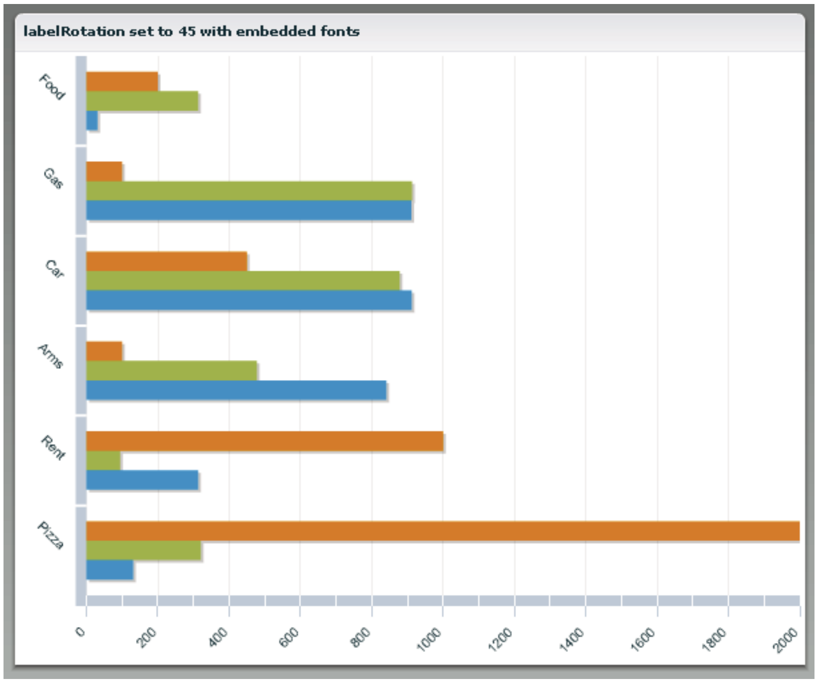
Formatting axis labels with the AxisRenderer class

The appearance of chart axis labels is defined by the AxisRenderer class.

Rotating chart axis labels

You can rotate axis labels by using the `labelRotation` property of the AxisRenderer object. You specify a number from -90 to 90, in degrees. If you set the `labelRotation` property to `null`, Flex determines an optimal angle and renders the axis labels.

The following example shows both sets of axis labels rotated 45 degrees:



To rotate axis labels, you must embed the font in the Flex application. If you rotate the the axis labels without embedding a font, they are rendered horizontally. You must also be sure to set the `embedAsCFF` CSS property to `false` because chart controls are MX controls and do not use the Spark text rendering engine. The default value of this property is `true`.

The following `<fx:Style>` block embeds a font in the Flex application, and then applies that font to the chart control that rotates its horizontal and vertical axis labels 45 degrees:

```
<?xml version="1.0"?>
<!-- charts/RotateAxisLabels.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";
     @font-face{
        src: url("../assets/MyriadWebPro.ttf");
        fontFamily: myMyriad;
        embedAsCFF: false;
     }
     mx|ColumnChart {
        fontFamily: myMyriad;
        fontSize: 20;
     }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Rotated Axis Labels">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="column"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis id="a1"
                categoryField="month"
                title="FY 2010"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
```

```
            <mx:LinearAxis id="a2"/>
         </mx:verticalAxis>
         <mx:horizontalAxisRenderers>
            <mx:AxisRenderer labelRotation="-90" axis="{a1}"/>
         </mx:horizontalAxisRenderers>
         <mx:verticalAxisRenderers>
            <mx:AxisRenderer labelRotation="45" axis="{a2}"/>
         </mx:verticalAxisRenderers>
         <mx:series>
            <mx:ColumnSeries displayName="Profit"
                 xField="month"
                 yField="profit"/>
            <mx:ColumnSeries displayName="Expenses"
                 xField="month"
                 yField="expenses"/>
         </mx:series>
      </mx:ColumnChart>
      <mx:Legend dataProvider="{column}"/>
   </s:Panel>
</s:Application>
```

Using a custom AxisRenderer object

To change the appearance of the axis labels, you can use the `labelRenderer` property of the AxisRenderer class. This lets you specify a class that defines the appearance of the label. The class must extend UIComponent and implement the IDataRenderer and IFlexDisplayObject interfaces. Its `data` property will be the label used in the chart. Typically, you write an ActionScript class that extends the ChartLabel class for the label renderer.

The following example defines a custom component inline by using the `<fx:Component>` tag. It adds ToolTip objects to the labels along the chart's horizontal axis so that the values of the longer labels are not truncated.

```
<?xml version="1.0"?>
<!-- charts/LabelRendererWithToolTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="setupDP()"
    width="550" height="600">

    <fx:Script>
     <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.ColumnChart;

        [Bindable]
        private var ac:ArrayCollection;

        public function setupDP():void{
            ac =  new ArrayCollection([
              [ "Label 1 is short.", 200000],
              [ "Label 2 is a fairly long label.", 150000],
              [ "Label 3 is an extremely long label. It contains 95 characters " +
                "and will likely be truncated.", 40000]
            ]);
        }
     ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="bc1"
            showDataTips="true"
            dataProvider="{ac}">
            <mx:series>
                <mx:ColumnSeries xField="0" yField="1"/>
            </mx:series>
            <mx:verticalAxis>
                <mx:LinearAxis id="va1"/>
            </mx:verticalAxis>
            <mx:horizontalAxis >
                <mx:CategoryAxis id="ha1"
                    dataProvider="{ac}"
                    categoryField="0"/>
            </mx:horizontalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer axis="{ha1}" canDropLabels="false">
                    <mx:labelRenderer>
                        <fx:Component>
                            <mx:Label toolTip="{this.myTip}">
                                <fx:Script><![CDATA[
                                    [Bindable]
                                    private var myTip:String;
                                    override public function set data(value:Object):void{
                                        if(value == null)
                                            return;
                                        myTip = value.text;
                                        var length:int = value.text.toString().length;
                                        if (length > 20) {
                                          text = value.text.toString().substr(0, 20) + "...";
                                        } else {
                                            text = value.text;
                                        }
                                    }
                                ]]></fx:Script>
                            </mx:Label>
                        </fx:Component>
                    </mx:labelRenderer>
                </mx:AxisRenderer>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer axis="{va1}" canDropLabels="false"/>
            </mx:verticalAxisRenderers>
        </mx:ColumnChart>
        <s:Label id="l1"
            color="white"
           text="Hover over the horizontal axis's labels to see the entire title rendered as
a ToolTip."/>
    </s:Panel>
</s:Application>
```

For an example of an ActionScript class that extends ChartLabel, see "Adding axis titles" on page 1262.

Skinning ChartItem objects

A ChartItem object represents a data point in a series. There is one ChartItem instance for each item in the series' data provider. ChartItem objects contain details about the data for the data point as well as the renderer (or *skin*) to use when rendering that data point in the series. The ChartItem renderers define objects such as the icon that represents a data point in a PlotChart control or the box that makes up a bar in a BarChart control.

Each series has a default renderer that Flex uses to draw that series' ChartItem objects. You can specify a new renderer to use with the series' `itemRenderer` style property. This property points to a class that defines the appearance of the ChartItem object.

The following table lists the available renderer classes for the ChartItem objects of each chart type:

Chart type	Available renderer classes
AreaChart	AreaRenderer
BarChart BubbleChart ColumnChart PlotChart	BoxItemRenderer CircleItemRenderer CrossItemRenderer DiamondItemRenderer ShadowBoxItemRenderer TriangleItemRenderer
CandlestickChart HLOCChart	CandlestickItemRenderer HLOCItemRenderer
LineChart	LineRenderer ShadowLineRenderer
PieChart	WedgeItemRenderer

The appearance of most renderers is self-explanatory. The BoxItemRenderer class draws ChartItem objects in the shape of boxes. The DiamondItemRenderer class draws ChartItem objects in the shape of diamonds. The ShadowBoxItemRenderer and ShadowLineRenderer classes add shadows to the ChartItem objects that they draw.

You can use existing classes to change the default renderers of chart items. The DiamondItemRenderer class is the default renderer for ChartItem objects in a data series in a PlotChart control. The following example uses the default DiamondItemRenderer class for the first data series. The second series uses the CircleItemRenderer class, which draws a circle to represent the data points in that series. The third series uses the CrossItemRenderer class, which draws a cross shape to represent the data points in that series.

```
<?xml version="1.0"?>
<!-- charts/PlotRenderers.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PlotChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <!-- First series uses default renderer. -->
           <mx:PlotSeries
                xField="expenses"
                yField="profit"
                displayName="Plot 1"/>
           <!-- Second series uses CircleItemRenderer. -->
           <mx:PlotSeries
                xField="amount"
                yField="expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.CircleItemRenderer"/>
           <!-- Third series uses CrossItemRenderer. -->
           <mx:PlotSeries
                xField="profit"
                yField="amount"
                displayName="Plot 3"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To apply a renderer to a series in ActionScript, you use the `setStyle()` method. In that method, you create a new ClassFactory and pass the renderer to its constructor. Flex generates an instance of this class to be the renderer. Be sure to import the appropriate classes when using renderer classes.

The following example sets the renderer for the second series to the CircleItemRenderer and the renderer for the third series to the CrossItemRenderer in ActionScript.

```
<?xml version="1.0"?>
<!-- charts/PlotRenderersAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();initSeriesStyles();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.charts.renderers.*;
    private function initSeriesStyles():void {
        // Second series uses CircleItemRenderer.
        series2.setStyle("itemRenderer", new
ClassFactory(mx.charts.renderers.CircleItemRenderer));
        // Third series uses CrossItemRenderer.
        series3.setStyle("itemRenderer", new
ClassFactory(mx.charts.renderers.CrossItemRenderer));
    }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PlotChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
```

```
                <mx:PlotSeries
                     id="series1"
                     xField="expenses"
                     yField="profit"
                     displayName="Plot 1"/>
                <mx:PlotSeries
                     id="series2"
                     xField="amount"
                     yField="expenses"
                     displayName="Plot 2"/>
                <mx:PlotSeries
                     id="series3"
                     xField="profit"
                     yField="amount"
                     displayName="Plot 3"/>
             </mx:series>
          </mx:PlotChart>
          <mx:Legend dataProvider="{myChart}"/>
     </s:Panel>
</s:Application>
```

Using multiple renderer classes

You can sometimes choose from more than one renderer for a chart series, depending on the series. These renderers let you change the appearance of your charts by adding shadows or graphics to the chart items.

Some series types require multiple renderers to completely render their data. For example, a LineSeries object has both an `itemRenderer` style property and a `lineSegmentRenderer` style property. The `itemRenderer` property specifies the renderer for the data items. The `lineSegmentRenderer` specifies the appearance of the line segments between items.

The other series type that requires two renderers is the AreaSeries. The `areaRenderer` property specifies the appearance of the area, and the `itemRenderer` specifies the appearance of the data items.

You can also specify the renderer to use for legends. The default is the class that the series' `itemRenderer` property specifies. For more information, see "Formatting Legend controls" on page 1241.

You can use multiple types of data series in a single chart. For example, you can use a ColumnSeries and a LineSeries to show something like a moving average over a stock price. In this case, you can use all the renderers supported by those series in the same chart. For more information on using multiple series, see "Using multiple data series" on page 1163.

Creating custom renderers

You can replace the `itemRenderer` property of a chart series with a custom renderer. You define the renderer on the `itemRenderer` style property for the chart series. This renderer can be a graphical renderer or a class that programmatically defines the renderer.

Creating graphical renderers

You can use a graphic file such as a GIF or JPEG to be used as a renderer on the chart series. You do this by setting the value of the `itemRenderer` style property to be an embedded image. This method of graphically rendering chart items is similar to the graphical skimming method used for other components, as described in "Creating graphical skins for MX components" on page 1671.

The following example uses the graphic file to represent data points on a PlotChart control:

```
<?xml version="1.0"?>
<!-- charts/CustomPlotRenderer.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PlotChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <!-- First series uses embedded image for renderer. -->
           <mx:PlotSeries
                xField="expenses"
                yField="profit"
                displayName="Plot 1"
                itemRenderer="@Embed(source='../assets/butterfly.gif')"
                radius="20"
                legendMarkerRenderer="@Embed(source='../assets/butterfly.gif')"/>
           <!-- Second series uses CircleItemRenderer. -->
           <mx:PlotSeries
                xField="amount"
                yField="expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.CircleItemRenderer"/>
           <!-- Third series uses CrossItemRenderer. -->
           <mx:PlotSeries
                xField="profit"
                yField="amount"
                displayName="Plot 3"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

This example uses the butterfly.gif graphic to represent each data point on the plot chart. It controls the size of the embedded image by using the `radius` style property.

You are not required to set the value of the `itemRenderer` property inline. You can also embed a graphic file in ActionScript as a Class, pass it to the ClassFactory class's constructor, and then reference it inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CustomPlotRendererAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.core.BitmapAsset;

     [Bindable]
     [Embed(source="../assets/butterfly.gif")]
     public var myButterfly:Class;
     [Bindable]
     public var myButterflyFactory:ClassFactory = new ClassFactory(myButterfly);
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PlotChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <!-- First series uses custom class renderer. -->
           <mx:PlotSeries
                id="series1"
                xField="expenses"
                yField="profit"
                displayName="Plot 1"
                itemRenderer="{myButterflyFactory}"
                legendMarkerRenderer="{myButterflyFactory}"
                radius="20"/>
           <!-- Second series uses CircleItemRenderer. -->
           <mx:PlotSeries
                id="series2"
                xField="amount"
                yField="expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.CircleItemRenderer"/>
           <!-- Third series uses CrossItemRenderer. -->
           <mx:PlotSeries
                id="series3"
                xField="profit"
                yField="amount"
                displayName="Plot 3"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also use the `setStyle()` method to apply the custom class to the item renderer. The following example sets the `itemRenderer` and `legendMarkerRenderer` style properties to the embedded image:

```
<?xml version="1.0"?>
<!-- charts/CustomPlotRendererStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();setStylesInit();"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.core.BitmapAsset;

     [Bindable]
     [Embed(source="../assets/butterfly.gif")]
     public var myButterfly:Class;
    private function setStylesInit():void {
        series1.setStyle("itemRenderer", new ClassFactory(myButterfly));
        series1.setStyle("legendMarkerRenderer", new ClassFactory(myButterfly));
    }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PlotChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <!-- First series uses custom class renderer. -->
           <mx:PlotSeries
                id="series1"
                xField="expenses"
                yField="profit"
                displayName="Plot 1"
                radius="20"/>
           <!-- Second series uses CircleItemRenderer. -->
           <mx:PlotSeries
                id="series2"
                xField="amount"
                yField="expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.CircleItemRenderer"/>
           <!-- Third series uses CrossItemRenderer. -->
           <mx:PlotSeries
                id="series3"
                xField="profit"
                yField="amount"
                displayName="Plot 3"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Creating programmatic renderers

Creating a custom renderer class for your chart items can give you more control than creating simple graphical renderers. Using class-based renderers is very similar to using programmatic skins, as described in "Creating programmatic skins for MX components" on page 1674.

One approach to is to extend the ProgrammaticSkin class and implement the IDataRenderer interface. In this approach, you can provide all of the logic for drawing chart items in your custom class, and maintain the greatest control over its appearance. For example, you use methods in the Graphics class to draw and fill the rectangles of the bars in a BarChart control.

When you implement the IDataRenderer interface, you must define a setter and getter method to implement the `data` property. This `data` property is of the type of the series item. In the case of a ColumnSeries, it is a ColumnSeriesItem. Other item types include BarSeriesItem, BubbleSeriesItem, LineSeriesItem, and PlotSeriesItem.

In your class, you override the updateDisplayList() method with the logic for drawing the chart item as well as setting any custom properties. You should also call the `super.updateDisplayList()` method.

The following example renders the chart items and uses an Array of colors to color each column in the ColumnChart control differently:

```
// charts/CycleColorRenderer.as
package { // Empty package.

  import mx.charts.series.items.ColumnSeriesItem;
  import mx.skins.ProgrammaticSkin;
  import mx.core.IDataRenderer;
  import flash.display.Graphics;

  public class CycleColorRenderer extends mx.skins.ProgrammaticSkin
     implements IDataRenderer {

     private var colors:Array = [0xCCCC99,0x999933,0x999966];
     private var _chartItem:ColumnSeriesItem;

     public function CycleColorRenderer() {
        // Empty constructor.
     }

     public function get data():Object {
        return _chartItem;
     }
     public function set data(value:Object):void {
        _chartItem = value as ColumnSeriesItem;
        invalidateDisplayList();
     }
     override protected function
        updateDisplayList(unscaledWidth:Number,unscaledHeight:Number):void {
           super.updateDisplayList(unscaledWidth, unscaledHeight);
           var g:Graphics = graphics;
           g.clear();
           g.beginFill(colors[(_chartItem == null)? 0:_chartItem.index]);
           g.drawRect(0, 0, unscaledWidth, unscaledHeight);
           g.endFill();
     }
  } // Close class.
} // Close package.
```

In your Flex application, you use this class as the renderer by using the `itemRenderer` property of the ColumnSeries, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ProgrammaticRenderer.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="ColumnChart control with a programmatic ItemRenderer">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <fx:Array>
            <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"
                itemRenderer="CycleColorRenderer"/>
           </fx:Array>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
</s:Application>
```

For more information on overriding the `updateDisplayList()` method, see “Implementing the updateDisplayList() method” on page 1677.

Formatting Legend controls

The Legend control is a subclass of UIComponent. You can use all the standard style properties to format the Legend control. Also, the Legend control has properties (such as `labelPlacement`, `markerHeight`, and `markerWidth)` that you can use to format its appearance. For information on creating Legend controls, see “Using Legend controls” on page 1316.

The following example sets styles by using CSS on the Legend control:

```
<?xml version="1.0"?>
<!-- charts/FormattedLegend.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
  <fx:Style>
    @namespace mx "library://ns.adobe.com/flex/mx";
     mx|Legend {
        labelPlacement:left;
        markerHeight:30;
        markerWidth:30;
     }
  </fx:Style>
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart with Legend">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="Profit"/>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also change the appearance of the lines on the LegendItem object’s marker. You do this with the `stroke` property or the `legendMarkerRenderer` property. For more information, see “Using strokes with chart controls” on page 1192.

You can place a Legend control anywhere in your application, as long as the control has access to the scope of the chart's data. You can place the Legend control in your application without a container, inside the same container as the chart, or in its own container, such as a Panel container. The latter technique gives the Legend control a border and title bar, and lets you use the `title` attribute of the Panel to create a title for the Legend control, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendInPanel.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart with Legend in Panel">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries 
                xField="amount" 
                displayName="Amount (in $USD)"/>
           <mx:BarSeries 
                xField="expenses" 
                displayName="Expenses (in $USD)"/>
           <mx:BarSeries 
                xField="profit" 
                displayName="Profit (in $USD)"/>
        </mx:series>
     </mx:BarChart>
     <s:Panel title="Legend">
        <mx:Legend dataProvider="{myChart}"/>
     </s:Panel>
  </s:Panel>
</s:Application>
```

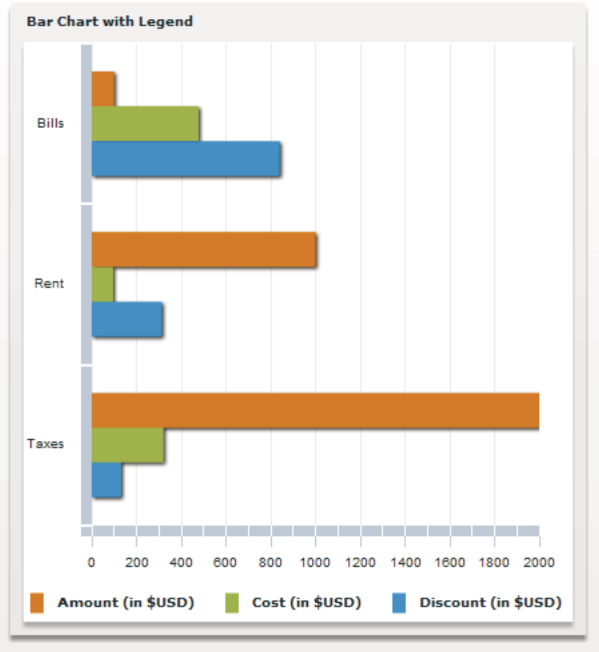
Setting the direction of legends

The `direction` property is a commonly used property to set on the Legend Control. This property of the `<mx:Legend>` tag causes the LegendItem objects to line up horizontally or vertically. The default value of `direction` is `vertical`; when you use this value, Flex stacks the LegendItem objects one on top of the other.

The following example sets the `direction` property to `horizontal`:

```
<?xml version="1.0"?>
<!-- charts/HorizontalLegend.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart with Legend">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries 
                xField="amount" 
                displayName="Amount (in $USD)"/>
           <mx:BarSeries 
                xField="expenses" 
                displayName="Expenses (in $USD)"/>
           <mx:BarSeries 
                xField="profit" 
                displayName="Profit (in $USD)"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}" direction="horizontal"/>
  </s:Panel>
</s:Application>
```

The following example shows the Legend with the `direction` property set to `horizontal`:



Formatting the legend markers

You can define the appearance of the legend markers by using a programmatic renderer class. Flex includes several default renderer classes that you can use for legend markers.

You can change the renderer of the LegendItem object from the default to one of the ChartItem renderers by using the series' `legendMarkerRenderer` style property. This property specifies the class to use when rendering the marker in all associated legends.

The following example overrides the default renderers and sets the legend markers of all three series to the DiamondItemRenderer class:

```
<?xml version="1.0"?>
<!-- charts/CustomLegendRenderer.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
```

```
  <s:Panel title="Plot Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PlotChart id="myChart"
          dataProvider="{srv.lastResult.data.result}"
          showDataTips="true">
        <mx:series>
           <!--
                Each series uses the default renderer for
                the ChartItems, but uses the DiamondItemRenderer
                for legend markers.
           -->
           <mx:PlotSeries
                xField="expenses"
                yField="profit"
                displayName="Plot 1"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"/>
           <mx:PlotSeries
                xField="amount"
                yField="expenses"
                displayName="Plot 2"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"/>
           <mx:PlotSeries
                xField="profit"
                yField="amount"
                displayName="Plot 3"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

If you do not explicitly set the `legendMarkerRenderer` property, the property uses the default class that the series' `itemRenderer` style property specifies. Each series has a default renderer that is used if neither of these style properties is specified.

You can create your own custom legend marker class. Classes used as legend marker renderers must implement the IFlexDisplayObject interface and, optionally, the ISimpleStyleClient and IDataRenderer interfaces.

For more information on available renderer classes, see "Skinning ChartItem objects" on page 1232.

Displaying data and labels in charts

Adobe® Flex® provides some techniques for displaying data and labels in your charts.

Working with axes

Each chart, except for a pie chart, has horizontal and vertical axes. There are two axis types: category and numeric. A *category axis* typically defines strings that represent groupings of items in the chart; for example, types of expenses (such as rent, utilities, and insurance) or names of employees. A *numeric axis* typically defines continuous data such as the amount of an expense or the productivity gains of the employee. These data define the height of a column, for example. Numeric axes include the LinearAxis, LogAxis, and DateTimeAxis.

You work with the axes objects to define their appearance, but also to define what data is displayed in the chart.

The following sections describe the CategoryAxis and NumericAxis classes.

About the CategoryAxis class

The CategoryAxis class maps discrete categorical data (such as states, product names, or department names) to an axis and spaces them evenly along it. This axis accepts any data type that can be represented by a String.

The `dataProvider` property of the CategoryAxis object defines the data provider that contains the text for the labels. In most cases, this can be the same data provider as the chart's data provider. A CategoryAxis object used in a chart inherits its `dataProvider` property from the containing chart, so you are not required to explicitly set the `dataProvider` property on a CategoryAxis object.

The `dataProvider` property of a CategoryAxis object can contain an Array of labels or an Array of objects. If the data provider contains objects, you use the `categoryField` property to point to the field in the data provider that contains the labels for the axis, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicColumn.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

If the data provider contains an Array of labels only, you do not specify the `categoryField` property.

You can customize the labels of the CategoryAxis object rather than use the axis labels in the data provider. You do this by providing a custom data provider. To provide a custom data provider, set the value of the CategoryAxis object's `dataProvider` property to a custom Array of labels, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CategoryAxisLabels.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     [Bindable]
     public var months:Array = [
        {Month:"January", monthAbbrev:"Jan"},
        {Month:"February", monthAbbrev:"Feb"},
        {Month:"March", monthAbbrev:"Mar"}
     ];
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
     <mx:AreaChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{months}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="profit"
                displayName="Profit"/>
           <mx:AreaSeries
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

For more information about chart data providers, see “Defining chart data” on page 1090.

You can also customize the labels using the `labelFunction` property of the CategoryAxis and NumericAxis classes. This property points to a callback function that refines the labels based on the existing data provider. For more information, see “Defining axis labels” on page 1269.

About the NumericAxis class

The NumericAxis class maps a set of continuous numerical values (such as sales volume, revenue, or profit) to coordinates on the screen. You do not typically use the NumericAxis base class directly. Instead, you use the following subclasses when you define your axis:

- LinearAxis
- LogAxis
- DateTimeAxis

These classes give you significant control over how to set the appearance and values of elements such as labels and tick marks along the axis.

You can use the `parseFunction` property to specify a custom method that formats the data points in your chart. This property is supported by all subclasses of the NumericAxis class. For a detailed description of using this property with the DateTimeAxis, see “Using the parseFunction property” on page 1253.

If you want to change the values of the labels, use the `labelFunction` property of the NumericAxis class. For more information, see “Defining axis labels” on page 1269.

About the LinearAxis subclass

The LinearAxis subclass is the simplest of the three NumericAxis subclasses. It maps numeric values evenly between minimum and maximum values along a chart axis. By default, Flex determines the minimum, maximum, and interval values from the charting data to fit all of the chart elements on the screen. You can also explicitly set specific values for these properties. The following example sets the minimum and maximum values to 40 and 50, respectively:

```
<?xml version="1.0"?>
<!-- charts/LinearAxisSample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="LineChart control with a linear axis">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:LineChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true"
            height="300" width="400">
            <mx:verticalAxis>
                <mx:LinearAxis
                    title="linear axis"
                    minimum="40"
                    maximum="50"
                    interval="1"/>
            </mx:verticalAxis>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="date"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:LineSeries
                    yField="close"
                    displayName="FRED close"/>
            </mx:series>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

About the LogAxis subclass

The LogAxis subclass is similar to the LinearAxis subclass, but it maps values to the axis logarithmically rather than linearly. You use a LogAxis object when the data in your chart has such a wide range that clusters of data points are lost to scale. LogAxis data also cannot be rendered if it is negative. For example, if you track the stock price of a successful company since 1929, it is useful to represent the data logarithmically rather than linearly so that the chart is readable.

When you use a LogAxis object, you set a multiplier that defines the values of the labels along the axis. You set the multiplier with the `interval` property. Values must be even powers of 10, and must be greater than or equal to 0. A value of 10 generates labels at 1, 10, 100, and 1000. A value of 100 generates labels at 1, 100, and 10,000. The default value of the `interval` property is 10. The LogAxis object rounds the interval to an even power of 10, if necessary.

As with the vertical and horizontal axes, you can also set the minimum and maximum values of a LogAxis object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LogAxisSample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_fe.send();srv_big.send()"
    height="600">
     <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    <fx:Declarations>
       <!-- View source of the following pages to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv_fe" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FE"/>
        <mx:HTTPService id="srv_big" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=BIG"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="LineChart control with a logarithmic axis">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:LineChart id="myChart"
            showDataTips="true"
            height="300" width="400">
            <mx:verticalAxis>
                <mx:LogAxis title="Log axis"
                    interval="10"
                    minimum="1"
                    maximum="5000"/>
            </mx:verticalAxis>
            <mx:horizontalAxis>
               <mx:DateTimeAxis id="h1" dataUnits="days"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:LineSeries dataProvider="{srv_fe.lastResult.data.result}"
                    yField="close"
                    displayName="FE close"/>
                <mx:LineSeries dataProvider="{srv_big.lastResult.data.result}"
                    yField="close"
                    displayName="BIG close"/>
            </mx:series>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

About the DateTimeAxis subclass

The DateTimeAxis subclass maps time-based values to a chart axis. The DateTimeAxis subclass calculates the minimum and maximum values to align with logical date and time units (for example, the nearest hour or the nearest week). The DateTimeAxis subclass also selects a time unit for the interval so that the chart renders a reasonable number of labels.

The `dataUnits` property of the DateTimeAxis subclass specifies how Flex should interpret the Date objects. Flex determines this property by default, but you can override it. To display data in terms of days, set the `dataUnits` property to *days*, as the following example shows:

```
<mx:DateTimeAxis dataUnits="days"/>
```

Valid values for the `dataUnits` property are `milliseconds`, `seconds`, `minutes`, `hours`, `days`, `weeks`, `months`, and `years`.

When assigning appropriate label units, a DateTimeAxis object does not assign any unit smaller than the units represented by the data. If the `dataUnits` property is set to `days`, the chart does not render labels for every hour, no matter what the minimum or maximum range is. To achieve this, you must set the value explicitly.

When using the DateTimeAxis class, you can filter out units when you set the `dataUnits` property to `days`. This lets you create a chart that shows a "work week" or some other configuration that omits certain days of the week. For more information, see “Omitting days on a DateTimeAxis object” on page 1257.

Some series use the value of the `dataUnits` property to affect their rendering. Specifically, most columnar series (such as Column, Bar, Candlestick, and HLOC controls) use the value of `dataUnits` to determine how wide to render their columns. If, for example, the ColumnChart control’s horizontal axis has its labels set to weeks and `dataUnits` set to days, the ColumnChart control renders each column at one-seventh the distance between labels.

About supported types

Data points on the DateTimeAxis object support the Date, String, and Number data types.

- Date: If the value of the data point is an instance of a Date object, it already represents an absolute date-time value and needs no interpretation. To pass a Date object as a data value, use the `parseFunction` property of the DateTimeAxis subclass. The `parseFunction` property returns a Date object. For more information, see “Using the parseFunction property” on page 1253.
- String: You can use any format that the `Date.parse()` method supports. The supported formats are:
  - *MM/YYYY* (for example, 02/2005)
  - *Day Month DD Hours:Minutes:Seconds GMT Year* (for example, Tue Feb 1 12:00:00 GMT-0800 2005)
  - *Day Month DD YYYY Hours:Minutes:Seconds AM|PM* (for example, Tue Feb 1 2005 12:00:00 AM)
  - *Day Month DD YYYY* (for example, Tue Feb 1 2005)
  - *MM/DD/YYYY* (for example, 02/01/2005)

  You can also write custom logic that uses the `parseFunction` property of the DateTimeAxis to take any data type and return a Date. For more information, see “Using the parseFunction property” on page 1253.
- Number: If you use a number, it is assumed to be the number of milliseconds since Midnight, 1/1/1970; for example, 543387600000. To get this value on an existing Date object, use the Date object’s `getTime()` method.

The following example specifies that the dates are displayed in units of days:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisSample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var deck:ArrayCollection = new ArrayCollection([
        {date:"08/01/2005", close:42.71},
        {date:"08/02/2005", close:42.99},
        {date:"08/03/2005", close:42.65}
     ]);
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Sample DateTimeAxis">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{deck}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="close"
                xField="date"
                displayName="DECK"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Using the parseFunction property

You use the `parseFunction` property of the DateTimeAxis object to specify a method that customizes the value of the data points. With this property, you specify a method that accepts a value and returns a Date object. The Date object is then used in the DateTimeAxis object of the chart. This lets you provide customizable input strings and convert them to Date objects, which Flex can then interpret for use in the DateTimeAxis.

The parsing method specified by the `parseFunction` property is called every time a value for the DateTimeAxis must be calculated. It is called each time a data point is encountered when the user interacts with the chart. Consequently, Flex might call the parsing method often, which can degrade an application's performance. Therefore, you should try to keep the amount of code in the parsing method to a minimum.

Flex passes only one parameter to the parsing method. This parameter is the value of the data point that you specified for the series. Typically, it is a String representing some form of a date. You cannot override this parameter or add additional parameters.

The following example shows a parsing method that creates a Date object from String values in the data provider that match the “YYYY, MM, DD” pattern:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisParseFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
  <fx:Script>
    <![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var ABC:ArrayCollection = new ArrayCollection([
        {date:"2005, 8, 1", close:42.71},
        {date:"2005, 8, 2", close:42.99},
        {date:"2005, 8, 3", close:44}
     ]);
     public function myParseFunction(s:String):Date {
        // Get an array of Strings from the
        // comma-separated String passed in.
        var a:Array = s.split(",");
        // Trace out year, month, and day values.
        trace("y:" + a[0]);
        trace("m:" + a[1]);
        trace("d:" + a[2]);
        // To create a Date object, you pass "YYYY,MM,DD",
        // where MM is zero-based, to the Date() constructor.
        var newDate:Date = new Date(a[0],a[1]-1,a[2]);
        return newDate;
     }
    ]]>
  </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="DateTimeAxis with parseFunction">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
    <mx:LineChart id="myChart"
        dataProvider="{ABC}"
        showDataTips="true">
     <mx:horizontalAxis>
        <mx:DateTimeAxis
            dataUnits="days"
            parseFunction="myParseFunction"/>
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries
            yField="close"
            xField="date"
            displayName="ABC"/>
     </mx:series>
    </mx:LineChart>
    <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Formatting DateTimeAxis labels

When assigning the units to display along the axis, the DateTimeAxis object uses the largest unit allowed to render a reasonable number of labels. The following table describes the default label format and the minimum range for each unit type:

Unit	Label format	Minimum range
Years	YYYY	If the minimum and maximum values span at least 2 years.
Months	MM/YY	Spans at least 2 months.
Weeks	DD/MM/YY	Spans at least 2 weeks.
Days	DD/MM/YY	Spans at least 1 day.
Hours	HH:MM	Spans at least 1 hour.
Minutes	HH:MM	Spans at least 1 minute.
Seconds	HH:MM:SS	Spans at least 1 second.
Milliseconds	HH:MM:SS:mmmm	Spans at least 1 millisecond.

You can restrict the list of valid units for a particular chart instance to a subset that makes sense for the use case. As with a LinearAxis object, you can specify minimum, maximum, and interval values for a DateTimeAxis object.

When rounding off values, the DateTimeAxis object determines if values passed to it should be displayed in the local time zone or UTC. You can set the `displayLocalTime` property to `true` to instruct the DateTimeAxis object to treat values as local time values. The default value is `false`.

To change the values of the labels, use the `labelFunction` property of the DateTimeAxis object. This property is inherited from the NumericAxis class and is described in “Defining axis labels” on page 1269.

Setting minimum and maximum values on a DateTimeAxis

You can define the range of values that any axis uses by setting the values of the `minimum` and `maximum` properties on that axis. For the DateTimeAxis class, however, you must use Date objects and not Numbers or Strings to define that range. To do this, you create bindable Date objects and bind the values of the `minimum` and `maximum` properties to those objects.

When creating Date objects, remember that the month parameter in the constructor is zero-based. The following example sets the minimum date for the axis to the first day of December 2006, and the maximum date for the axis to the first day of February 2007. The result is that Flex excludes the first and last data points in the ArrayCollection because those dates fall outside of the range set on the axis:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisRange.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     // To create a Date object, you pass "YYYY,MM,DD",
     // where MM is zero-based, to the Date() constructor.
     [Bindable]
     public var minDate:Date = new Date(2006, 11, 1);
     [Bindable]
     public var maxDate:Date = new Date(2007, 1, 1);
     [Bindable] public var myData:ArrayCollection = new
     ArrayCollection([
        {date: "11/03/2006", amt: 12345},
        {date: "12/02/2006", amt: 54331},
        {date: "1/03/2007", amt: 34343},
        {date: "2/05/2007", amt: 40299}
     ]);
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="DateTimeAxis with range">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{myData}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:DateTimeAxis
                dataUnits="months"
                minimum="{minDate}"
                maximum="{maxDate}"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="amt"
                xField="date"
                displayName="My Data"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also represent the range of dates in MXML by using the following syntax:

```
<mx:horizontalAxis>
    <mx:DateTimeAxis dataUnits="months">
        <mx:minimum>
            <mx:Date fullYear="2005" month="11" date="1"/>
        </mx:minimum>
        <mx:maximum>
            <mx:Date fullYear="2007" month="1" date="1"/>
        </mx:maximum>
    </mx:DateTimeAxis>
</mx:horizontalAxis>
```

Omitting days on a DateTimeAxis object

You can exclude particular days or ranges of days from a chart. This lets you create charts that show only the days of the work week or that exclude other days of the week for other reasons.

For example, if you create a LineChart control that shows a stock price over the course of an entire month, the source data typically includes pricing data only for Monday through Friday. Values for the weekend days are typically not in the data. So, the chart control extrapolates values by extending the line through the weekend days on the chart, which makes it appear as though there is data for those days. If you *disable* the weekend days, the chart control removes those days from the chart and the line draws only the days that are not disabled. There is no breakage or other indicator that there are omitted days.

To disable days of the week or ranges of days in your charts, you must set the `dataUnits` property of the DateTimeAxis object to `days`. You then use the `disabledDays` or `disabledRanges` properties of the DateTimeAxis object.

The value of the `disabledDays` property of DateTimeAxis is an Array of numbers. These numbers correspond to days of the week, with 0 being Sunday, 1 being Monday, and so on, up until 6 being Saturday.

The following example excludes Saturdays and Sundays from the chart by setting the value of the `disabledDays` property to an Array that contains 0 and 6:

```
<?xml version="1.0"?>
<!-- charts/WorkWeekAxis.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
       <!-- View source of the following pages to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
    </fx:Declarations>
  <fx:Script><![CDATA[
     /* Create an Array that specifies which days to exclude.
        0 is Sunday and 6 is Saturday. */
     [Bindable]
     private var offDays:Array = [0,6];

  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="WorkWeekAxis Example">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"
            disabledDays="{offDays}"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="close" xField="date"
                displayName="FRED"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To exclude a range of dates from the DateTimeAxis object, you use the `disabledRanges` property. This property takes an of objects. Each object specifies two dates: a `rangeStart` and `rangeEnd` property. The following example excludes August 13, and then the range of days between August 27 and August 31:

```
<?xml version="1.0"?>
<!-- charts/DisabledDateRanges.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init()"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var deck:ArrayCollection = new ArrayCollection([
        {date:"08/01/2007", close:42},
        {date:"08/02/2007", close:43},
        {date:"08/03/2007", close:43},
        {date:"08/06/2007", close:42},
        {date:"08/07/2007", close:38},
        {date:"08/08/2007", close:37},
        {date:"08/09/2007", close:39},
        {date:"08/10/2007", close:41},
        {date:"08/13/2007", close:45},
        {date:"08/14/2007", close:47},
        {date:"08/15/2007", close:48},
        {date:"08/16/2007", close:42},
        {date:"08/17/2007", close:43},
        {date:"08/20/2007", close:45},
        {date:"08/21/2007", close:50},
        {date:"08/22/2007", close:51},
        {date:"08/23/2007", close:55},
        {date:"08/24/2007", close:51},
        {date:"08/27/2007", close:49},
        {date:"08/28/2007", close:51},
        {date:"08/29/2007", close:50},
        {date:"08/30/2007", close:49},
        {date:"08/31/2007", close:54}
     ]);

     private function myParseFunction(s:String):Date {
        var a:Array = s.split("/");
        var newDate:Date = new Date(a[2],a[0]-1,a[1]);
        return newDate;
     }
     private var d1:Date, d2:Date, d3:Date;
     [Bindable]
     private var offRanges:Array = new Array ([]);
     private function init():void {
        d1 = new Date("08/13/2007");
        d2 = new Date("08/27/2007");
        d3 = new Date("08/31/2007");
        offRanges = [ {rangeStart:d1, rangeEnd:d1},{rangeStart:d2, rangeEnd:d3} ];
     }

     private var series1:LineSeries;
  ]]></fx:Script>

    <s:layout>
```

```
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Disabled Date Ranges">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:LineChart id="myChart"
        dataProvider="{deck}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:DateTimeAxis
            dataUnits="days"
            parseFunction="myParseFunction"
            disabledRanges="{offRanges}"/>
        </mx:horizontalAxis>

        <mx:series>
           <mx:LineSeries id="mySeries"
                yField="close"
                xField="date"
                displayName="DECK"/>
        </mx:series>
    </mx:LineChart>
    <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The following example expands on the previous example, except it adds a DateChooser control. You can select days on the DateChooser that are then removed from the chart.

```
<?xml version="1.0"?>
<!-- charts/DisabledDateRangesWithDateChooser.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init()"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var deck:ArrayCollection = new ArrayCollection([
        {date:"08/01/2007", close:42},
        {date:"08/02/2007", close:43},
        {date:"08/03/2007", close:43},
        {date:"08/06/2007", close:42},
        {date:"08/07/2007", close:38},
        {date:"08/08/2007", close:37},
        {date:"08/09/2007", close:39},
        {date:"08/10/2007", close:41},
        {date:"08/13/2007", close:45},
        {date:"08/14/2007", close:47},
        {date:"08/15/2007", close:48},
        {date:"08/16/2007", close:42},
        {date:"08/17/2007", close:43},
        {date:"08/20/2007", close:45},
        {date:"08/21/2007", close:50},
        {date:"08/22/2007", close:51},
        {date:"08/23/2007", close:55},
        {date:"08/24/2007", close:51},
        {date:"08/27/2007", close:49},
        {date:"08/28/2007", close:51},
        {date:"08/29/2007", close:50},
        {date:"08/30/2007", close:49},
        {date:"08/31/2007", close:54}
     ]);
     // Define weekend days to be removed from chart.
     [Bindable]
     private var offDays:Array = [0,6];

     [Bindable]
     private var dateChooserDisabledRanges:Array = [];

     private function init():void {
        // Limit selectable range to August of 2007 on DateChooser.
        dateChooserDisabledRanges = [
            {rangeEnd: new Date(2007, 6, 31)},
            {rangeStart: new Date(2007, 8, 1)}
        ];
        // Disable weekend days on DateChooser.
        dc1.disabledDays = [0, 6];
     }

     [Bindable]
     private var offRanges:Array = new Array([]);
     private function onDateChange(e:Event):void {
        // Get the start and end date of the range.
        var startDate:Date = e.currentTarget.selectedRanges[0].rangeStart;
        var endDate:Date = e.currentTarget.selectedRanges[0].rangeEnd;
        var d:Object = {rangeStart:startDate, rangeEnd:endDate};

        // Add object to list of ranges to disable on chart.
        offRanges.push(d);

        // Refresh the chart series with the new offRanges.
        var mySeries:Array = [];
        mySeries.push(series1);
        myChart.series = mySeries;
        // Show the current ranges.
        ta1.text = "";
        for (var i:int = 0; i < offRanges.length; i++) {
            for (var s:String in offRanges[i]) {
                ta1.text += s + ":" + offRanges[i][s] + "\n";
            }
        }
     }
     private function clearAllDisabledDates():void {
        offRanges = [];
        dc1.selectedDate = null;
        ta1.text = "";
     }

  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Disabled date ranges">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:LineChart id="myChart"
       dataProvider="{deck}"
       showDataTips="true">
       <mx:horizontalAxis>
          <mx:DateTimeAxis
             id="dtAxis"
             dataUnits="days"
             disabledDays="{offDays}"
             disabledRanges="{offRanges}"/>
       </mx:horizontalAxis>
       <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="60"/>
       </mx:verticalAxis>
       <mx:series>
          <mx:LineSeries id="series1"
               yField="close"
               xField="date"
               displayName="DECK"/>
       </mx:series>
    </mx:LineChart>
    <mx:DateChooser id="dc1"
        showToday="false"
        click="onDateChange(event)"
        displayedMonth="7"
        displayedYear="2007"
        disabledRanges="{dateChooserDisabledRanges}"/>
    <mx:Legend dataProvider="{myChart}"/>

    <s:Button id="b1" label="Refresh" click="clearAllDisabledDates()"/>
    <s:TextArea id="ta1" width="600" height="400"/>
  </s:Panel>
</s:Application>
```

Adding axis titles

Each axis in a chart control can include a title that describes the purpose of the axis to the users. Flex does not add titles to the chart's axes unless you explicitly set them. To add titles to the axes of a chart, you use the `title` property of the axis object. This is CategoryAxis or one of the NumericAxis subclasses such as DateTimeAxis, LinearAxis, or LogAxis. To set a style for the axis title, use the `axisTitleStyleName` property of the chart control.

The following example sets the titles of the horizontal and vertical axes (in MXML and ActionScript), and applies the styles to those titles:

```
<?xml version="1.0"?>
<!-- charts/AxisTitles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();setTitles();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script><![CDATA[
        private function setTitles():void {
            la1.title="Dollars";
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        .myStyle {
            fontFamily:Verdana;
            fontSize:12;
            color:#4691E1;
            fontWeight:bold;
            fontStyle:italic;
        }
    </fx:Style>
    <s:Panel title="Axis with title">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart" 
            showDataTips="true"
            axisTitleStyleName="myStyle"
            dataProvider="{srv.lastResult.data.result}">
            <mx:verticalAxis>
                <mx:LinearAxis id="la1"/>
            </mx:verticalAxis>
            <mx:horizontalAxis>
                <mx:CategoryAxis title="FY 2009" categoryField="month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries
                    xField="month"
                    yField="profit"
                    displayName="Profit"/>
                <mx:ColumnSeries
                    xField="month"
                    yField="expenses"
                    displayName="Expenses"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

You can also use embedded fonts for your axis titles. The following example embeds the font and sets the style for the vertical axis title:

```
<?xml version="1.0"?>
<!-- charts/AxisTitleEmbedFont.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
  <fx:Declarations>
     <!-- View source of the following page to see the structure of the data that Flex uses in
this example. -->
     <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
     <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
  </fx:Declarations>
  <fx:Style>
     @font-face{
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily:myMyriad;
     }

     .myEmbeddedStyle {
        fontFamily:myMyriad;
        fontSize:20;
     }
  </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Axis title with embedded font">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="column"
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}"
        axisTitleStyleName="myEmbeddedStyle">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"
                title="FY 2009"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

For information on embedding fonts, see “Embed fonts” on page 1571.

You can take advantage of the fact that the chart applies the `axisTitleStyleName` property without explicitly specifying it, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CSSAxisTitle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";
     .axisTitles {
        color:red;
        fontWeight:bold;
        fontFamily:Arial;
        fontSize:20;
     }
     mx|ColumnChart {
        axisTitleStyleName:axisTitles;
     }
  </fx:Style>
```

```
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Styling Axis Titles with CSS">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="column"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month" title="FY 2009"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </s:Panel>
</s:Application>
```

You can also apply the title style to the axis, as the following example shows:

```
<mx:CategoryAxis title="State" styleName="myEmbeddedStyle"/>
```

To change the appearance of the title on a chart, you can also use the `titleRenderer` property of the AxisRenderer class. This lets you specify a class that defines the appearance of the title. The class must extend UIComponent and implement the IDataRenderer and IFlexDisplayObject interfaces. Its `data` property will be the title used in the chart.

Typically, you extend the ChartLabel class to create a custom title renderer. In that class, you override the `updateDisplayList()` method. In the `updateDisplayList()` method, you define the appearance of the box surrounding the title as well as the appearance of the title's text.

The following example is a custom title renderer. It creates a gradient fill background for each of the axis titles.

```
// charts/MyTextRenderer.as
package {
    import mx.charts.chartClasses.ChartLabel;
    import mx.charts.*;
    import flash.display.*;
    import flash.geom.Matrix;

    public class MyTextRenderer extends ChartLabel {
        public function MyTextRenderer() {
            super();
        }

        override protected function updateDisplayList(w:Number, h:Number):void {
            super.updateDisplayList(w, h);

            this.setStyle("textAlign","center");
            var g:Graphics = graphics;
            g.clear();
            var m:Matrix = new Matrix();
            m.createGradientBox(w+100,h,0,0,0);
            g.beginGradientFill(GradientType.LINEAR,[0xFF0000,0xFFFFFF],
                [.1,1],[0,255],m,null,null,0);
            g.drawRect(-50,0,w+100,h);
            g.endFill();
        }
    }
}
```

To use this class in your application, you point the `titleRenderer` property to it. This assumes that the class is in your source path. In this case, store both the MyTextRenderer.as and RendererSample.mxml files in the same directory.

The following example applies the custom title renderer to all axis titles.

```
<?xml version="1.0"?>
<!-- charts/RendererSample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <s:Panel title="Axis titles with custom text renderers">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
         <mx:ColumnChart id="bc1"
             showDataTips="true"
             dataProvider="{srv.lastResult.data.result}">
             <mx:series>
                 <mx:ColumnSeries xField="month" yField="profit"/>
             </mx:series>
             <mx:verticalAxis>
                 <mx:LinearAxis id="va1" title="Dollars"/>
             </mx:verticalAxis>
             <mx:horizontalAxis >
                 <mx:CategoryAxis id="ha1"
                     categoryField="month"
                     title="FY 2009"/>
             </mx:horizontalAxis>
             <mx:horizontalAxisRenderers>
                 <mx:AxisRenderer
                     axis="{ha1}"
                     canDropLabels="true"
                     titleRenderer="MyTextRenderer"/>
             </mx:horizontalAxisRenderers>
             <mx:verticalAxisRenderers>
                 <mx:AxisRenderer
                     axis="{va1}"
                     canDropLabels="true"
                     titleRenderer="MyTextRenderer"
                     verticalAxisTitleAlignment="vertical"/>
             </mx:verticalAxisRenderers>
         </mx:ColumnChart>
         <mx:Spacer width="50"/>
         <mx:BarChart id="bc2"
             showDataTips="true"
             dataProvider="{srv.lastResult.data.result}">
             <mx:series>
                 <mx:BarSeries xField="expenses" yField="month"/>
             </mx:series>
             <mx:horizontalAxis>
                 <mx:LinearAxis id="ha2" title="Dollars"/>
             </mx:horizontalAxis>
             <mx:verticalAxis>
```

```
                <mx:CategoryAxis id="va2"
                    categoryField="month"
                    title="FY 2009"/>
            </mx:verticalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
                    axis="{ha2}"
                    canDropLabels="false"
                    titleRenderer="MyTextRenderer"/>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer
                    axis="{va2}"
                    canDropLabels="true"
                    titleRenderer="MyTextRenderer"/>
            </mx:verticalAxisRenderers>
        </mx:BarChart>
    </s:Panel>

</s:Application>
```

Defining axis labels

You define the values of axis labels on the horizontal axis or vertical axis. You can customize these values by using the available data in the series or you can disable these values altogether.

Disabling axis labels

You can disable labels by setting the value of the `showLabels` property to `false` on the AxisRenderer object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/DisabledAxisLabels.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Disabled Axis Labels">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="column"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis id="a1" categoryField="month"/>
            </mx:horizontalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    showLabels="false"/>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer axis="{a1}" showLabels="false"/>
            </mx:verticalAxisRenderers>
            <mx:series>
                <mx:ColumnSeries
                    xField="month"
                    yField="profit"
                    displayName="Profit"/>
                <mx:ColumnSeries
                    xField="month"
                    yField="expenses"
                    displayName="Expenses"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{column}"/>
    </s:Panel>
</s:Application>
```

Note that any time you want to use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer’s `axis` property.

Customizing axis labels

You can customize the value of axis labels by using the `labelFunction` callback function of the axis. The function specified in `labelFunction` returns a String, Number, or Date object that Flex displays as the axis label.

The callback function signature for a NumericAxis object (including the DateTimeAxis, LinearAxis, and LogAxis classes) is:

```
function_name(labelValue:Object, previousLabelValue:Object, axis:IAxis):return_type
```

The callback function signature for a CategoryAxis object is:

```
function_name(labelValue:Object, previousLabelValue:Object, axis:axis_type,
labelItem:Object):return_type
```

The following table describes the parameters of the callback function:

Parameter	Description
labelValue	The value of the current label.
previousLabelValue	The value of the label preceding this label. If this is the first label, the value of ***previousLabelValue*** is `null`.

Parameter	Description
axis	The axis object, such as CategoryAxis or NumericAxis.

labelItem	A reference to the label object. This argument is only passed in for a CategoryAxis object. For NumericAxis subclasses such as LogAxis, DateTimeAxis, and LinearAxis objects, you omit this argument.
	This object contains a name/value pair for the chart data. For example, if the data provider defines the Month, Profit, and Expenses fields, this object might look like the following:
	`Profit:1500`
	`Month:Mar`
	`Expenses:500`
	You can access the values in this object by using dot-notation for dynamic objects, as the following example shows:
	`return "$" + labelItem.Profit;`

return_type	The type of object that the callback function returns. This can be any object type, but is most commonly a String for CategoryAxis axes, a Number for NumericAxis objects, or a Date object for DateTimeAxis objects.

When you use the `labelFunction`, you must be sure to import the class of the axis or the entire charts package; for example:

```
import mx.charts.*;
```

The following example defines a `labelFunction` for the horizontal CategoryAxis object. In that function, Flex appends ’10 to the axis labels, and displays the labels as Jan ’10, Feb ’10, and Mar ‘10. For the vertical axis, this example specifies that it is a LinearAxis, and formats the values to include a dollar sign and a thousands separator (by setting the `useGrouping` property to `true` on the NumberFormatter). The NumberFormatter also removes decimal values by setting the `fractionalDigits` property to 0. The return types of the label formatting functions are Strings.

```
<?xml version="1.0"?>
<!-- charts/CustomLabelFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
         <s:NumberFormatter id="numForm" useGrouping="true" fractionalDigits="0"/>
    </fx:Declarations>
  <fx:Script><![CDATA[
    import mx.charts.*;
    // This method customizes the values of the axis labels.
    // This signature (with 4 arguments) is for a CategoryAxis.
    public function defineLabel(cat:Object, 
                                pcat:Object,
                                ax:CategoryAxis,
                                labelItem:Object):String
    {
        // Show contents of the labelItem:
```

```
        for (var s:String in labelItem) {
            trace(s + ":" + labelItem[s]);
        }
        // Return the customized categoryField value:
        return cat + " '10";
        // Note that if you did not specify a categoryField,
        // cat would refer to the entire object and not the
        // value of a single field. You could then access
        // fields by using cat.field_name.
    }
    // For a NumericAxis, you do not use the labelItem argument.
    // This example uses a NumberFormatter to add a thousands
    // separator (by setting useGrouping to true) and removes decimal values
    // (by setting fractionalDigits to 0).
    public function defineVerticalLabel(
                                        cat:Object,
                                        pcat:Object,
                                        ax:LinearAxis):String
    {
        return "$" + numForm.format(cat);
    }
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Custom Label Function">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="column"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                categoryField="month"
                title="Expenses"
                labelFunction="defineLabel"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis title="Income"
                minimum="0" maximum="2500"
                labelFunction="defineVerticalLabel"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
</s:Application>
```

In the previous example, if you use a CategoryAxis but do not specify the value of the `categoryField` property on the axis, the label format function receives an object rather than a value for the first argument. In that case, you must drill down into the object to return a formatted String.

You can also customize labels by using the `labelFunction` property of the AxisRenderer class. This lets you control the labels if you use multiple axes. The callback function signature for the AxisRenderer's label function is:

```
function_name(axisRenderer:IAxisRenderer, label:String):String
```

Because this particular callback function takes an argument of type IAxisRenderer, you must import that class when you use this function:

```
import mx.charts.chartClasses.IAxisRenderer;
```

The following example specifies the value of the `labelFunction` property for one of the vertical axis renderers. The resulting function, `CMstoInches()`, converts centimeters to inches for the axis' labels.

```
<?xml version="1.0"?>
<!-- charts/CustomLabelsOnAxisRenderer.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

    <fx:Script>
    <![CDATA[
        import spark.formatters.NumberFormatter;
        import mx.charts.chartClasses.IAxisRenderer;
        import mx.collections.ArrayCollection;

        private function CMstoInches(ar:IAxisRenderer, strCMs:String):String {
            var n:NumberFormatter = new NumberFormatter();
            // Set precision to 1 decimal place:
            n.fractionalDigits = 1;
            return n.format((Number(strCMs) * 0.393700787).toString());
        }

        [Bindable]
        private var SampleHeightData:ArrayCollection = new ArrayCollection([
            { Age: "Birth", height: 53},
            { Age: "3", height: 57 },
            { Age: "6", height: 64 },
            { Age: "9", height: 70 },
            { Age: "12", height: 82 },
            { Age: "15", height: 88 }
        ]);

        [Bindable]
        private var HeightData:ArrayCollection = new ArrayCollection([
            { Age: "Birth", 5: 52, 10: 53, 25:54, 50:58, 75:60, 90:62, 95:63 },
            { Age: "3", 5: 56, 10: 57, 25:58, 50:62, 75:64, 90:66, 95:67 },
            { Age: "6", 5: 62, 10: 63, 25:64, 50:68, 75:70, 90:72, 95:73 },
            { Age: "9", 5: 66, 10: 67, 25:68, 50:72, 75:74, 90:76, 95:77 },
            { Age: "12", 5: 70, 10: 71, 25:72, 50:76, 75:80, 90:82, 95:83 },
            { Age: "15", 5: 74, 10: 75, 25:76, 50:80, 75:84, 90:86, 95:87 }
        ]);
    ]]>
```

```
</fx:Script>
<fx:Declarations>
    <mx:SolidColorStroke id="s1" weight="1"  />
</fx:Declarations>

<s:layout>
    <s:VerticalLayout/>
</s:layout>
<s:Panel title="Multiple Axis Example, Boys: Age - Height percentiles"
    height="100%" width="100%">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <mx:ColumnChart id="linechart" height="100%" width="100%"
        paddingLeft="5"
        paddingRight="5"
        showDataTips="true"
        dataProvider="{HeightData}">
        <mx:seriesFilters>
            <fx:Array/>
        </mx:seriesFilters>
        <mx:backgroundElements>
            <mx:GridLines gridDirection="both"/>
        </mx:backgroundElements>
        <mx:horizontalAxis>
            <mx:CategoryAxis id="h1"
                categoryField="Age"
                title="Age in Months"
                ticksBetweenLabels="false"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis id="v1"
                title="Height"
                baseAtZero="false"/>
        </mx:verticalAxis>
        <mx:verticalAxisRenderers>
            <mx:AxisRenderer
                axis="{v1}"
                placement="right"/>
            <mx:AxisRenderer
                axis="{v1}"
                placement="right"
                labelFunction="CMstoInches"
                highlightElements="true"/>
        </mx:verticalAxisRenderers>
        <mx:horizontalAxisRenderers>
```

```
                <mx:AxisRenderer axis="{h1}" placement="bottom"/>
                <mx:AxisRenderer axis="{h1}" placement="top"/>
            </mx:horizontalAxisRenderers>
            <mx:series>
                <mx:LineSeries yField="5"  form="curve" displayName="5%"/>
                <mx:LineSeries yField="10" form="curve" displayName="10%"/>
                <mx:LineSeries yField="25" form="curve" displayName="25%"/>
                <mx:LineSeries yField="50" form="curve" displayName="50%"/>
                <mx:LineSeries yField="75" form="curve" displayName="75%"/>
                <mx:LineSeries yField="90" form="curve" displayName="90%"/>
                <mx:LineSeries yField="95" form="curve" displayName="95%"/>
                <mx:ColumnSeries displayName="Height of Child X"
                    dataProvider="{SampleHeightData}"
                    yField="height"
                    fills="{[0xCC6600]}"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{linechart}"/>
    </s:Panel>
</s:Application>
```

Another way to customize the labels on an axis is to set a custom data provider for the labels. This can be done if you are using the CategoryAxis and not the NumericAxis class for the axis values. For more information, see "About the CategoryAxis class" on page 1247.

For the PieChart control, you can customize labels with the label function defined on the PieSeries class. For more information, see "Using data labels with PieChart controls" on page 1153.

Setting ranges on a NumericAxis

Flex determines the minimum and maximum values along an axis and sets the interval based on the settings of the NumericAxis object. You can override the values that Flex calculates. By changing the range of the data displayed in the chart, you also change the range of the tick marks.

The following table describes the properties of the axis that define the ranges along the axes:

Property	Description
`minimum`	The lowest value of the axis.
`maximum`	The highest value of the axis.
`interval`	The number of units between values along the axis.

The following example defines the range of the y-axis:

```
<?xml version="1.0"?>
<!-- charts/LinearAxisSample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="LineChart control with a linear axis">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:LineChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true"
            height="300" width="400">
            <mx:verticalAxis>
                <mx:LinearAxis
                    title="linear axis"
                    minimum="40"
                    maximum="50"
                    interval="1"/>
            </mx:verticalAxis>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="date"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:LineSeries
                    yField="close"
                    displayName="FRED close"/>
            </mx:series>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

In this example, the minimum value displayed along the y-axis is 10, the maximum value is 100, and the interval is 10. Therefore, the label text is 10, 20, 30, 40, and so on.

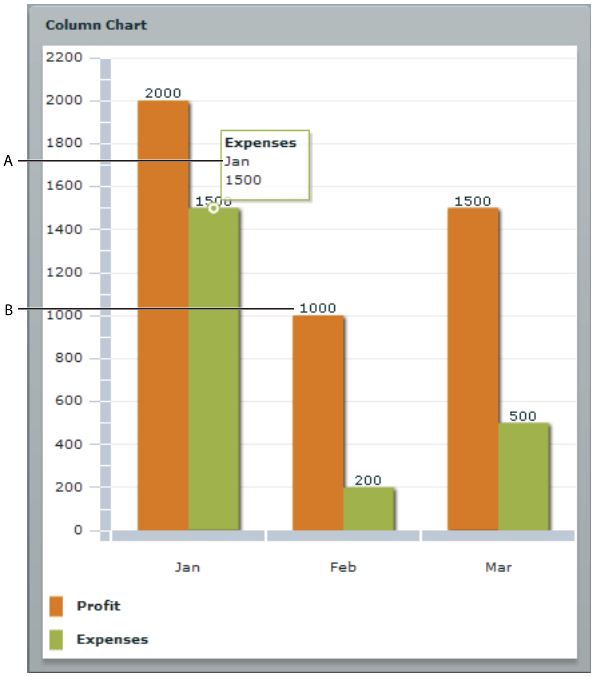
To set the minimum and maximum values on a DateTimeAxis, you must use Date objects rather than Strings or Numbers in the axis's tag. For more information, see "Setting minimum and maximum values on a DateTimeAxis" on page 1256.

For information about setting the length and location of tick marks, see "Formatting tick marks" on page 1187.

Using data labels

Data labels show static data on the chart control. They typically appear on the chart for each chart element (such as all pie wedges or all bars) for the entire time the chart is active. They are different from DataTip objects in that they typically show only the simple value of a chart element in a series (for example, a column's height or a pie wedge's percentage) and do not include other information such as the series name or complex formatting. DataTip controls are more interactive, since they appear and disappear as the user moves the mouse over chart elements. For more information about DataTip objects, see "Using DataTip objects" on page 1287.

The following example shows a column chart with an active DataTip object and visible data labels:



A. *DataTip* ***B.*** *Data label*

The following chart series support data labels:

- BarSeries
- ColumnSeries
- PieSeries

Just like DataTip controls, data labels are not enabled by default. You can add data labels by setting the value of the series' `labelPosition` property to `inside` or `outside` (for BarSeries and ColumnSeries) or `inside`, `outside`, `callout`, or `insideWithCallout` (for PieSeries). For more information, see "Adding data labels" on page 1279. The default value of the series's `labelPosition` property is `none`.

The following example enables data labels on the columns by setting the value of the `labelPosition` style property to `inside`. You can click the button to change the position of the labels to `outside`.

```
<?xml version="1.0"?>
<!-- charts/BasicDataLabel.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();setStartLabelLocation();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     private function setStartLabelLocation():void {
        cs1.setStyle("labelPosition", "inside");
        cs2.setStyle("labelPosition", "inside");
     }

     private function changeLabelLocation():void {
        var pos:String = cs1.getStyle("labelPosition");
        if (pos == "inside") {
            pos = "outside";
        } else {
            pos = "inside";
        }
        cs1.setStyle("labelPosition", pos);
        cs2.setStyle("labelPosition", pos);
     }

  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis minimum="0" maximum="2500"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="cs1"
                xField="month"
                yField="profit"
                displayName="Profit"/>
           <mx:ColumnSeries id="cs2"
                xField="month"
                yField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
  <s:Button id="b1" label="Change Label Location"
    click="changeLabelLocation()"/>
</s:Application>
```

Adding data labels

By default, BarSeries, ColumnSeries, and PieSeries objects do not display data labels. To enable data labels on BarSeries and ColumnSeries objects, set the value of the `labelPosition` property to `inside` or `outside`. To enabled data labels on PieSeries, set the value of the `labelPosition` property to `inside`, `outside`, `callout`, or `insideWithCallout`.

The default value of the `labelPosition` property is `none`.

Because `labelPosition` property is a style property, you can set it either inline in the series' tag or by using the `setStyle()` method, as the following example shows:

```
mySeries.setStyle("labelPosition", "outside");
```

The contents of the data label are determined by several factors, in order of precedence:

- `labelField`—Specifies a field in the data provider that sets the contents of the data label. This overrides any method specified by the `labelFunction` property.
- `labelFunction`—Specifies a method that takes several arguments and returns a String that the chart series uses for the data label's contents. For more information, see "Customizing data label values" on page 1283.
- Default—If neither the `labelField` nor the `labelFunction` are specified, the default content of the data label is the value that the series uses to draw the chart element. This is the `xField` value for a ColumnSeries and the `yField` value for a BarSeries. For a PieSeries, the default content of the data labels is the value of the `field` property.

Setting the `labelPosition` property to `inside` restricts the amount of styling you can perform on the data labels. If the data labels are inside the chart elements (for example, inside the column in a ColumnSeries), their size is restricted by the available space within that element. A data label cannot be bigger than the chart element that contains it. If you try to set a data label to be larger than its underlying chart element, Flex scales and possibly truncates the contents of the data label. As a result, you should usually set the value of the `labelPosition` property to `outside`. For information about styling your data labels, see "Styling data labels" on page 1280.

For 100%, stacked, and overlaid charts, the values of the series's `labelPosition` property can only be `inside`. If you set the `labelPosition` property to `outside` for these types of bar and column series, the value is ignored and the labels are rendered as if the `labelPosition` was `inside`.

If you set the `labelPosition` property to `inside`, you cannot rotate data labels.

Styling data labels

You can style the data labels so that their appearance fits into the style of your application. You can apply the following styles to data labels:

- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `labelAlign` (when the `labelPosition` property is set to `inside` only)
- `labelPosition`
- `labelSizeLimit`

You can apply these styles by using type or class selectors in CSS or by using the `setStyle()` method. You can also set these properties inline in the series's MXML tag.

The following example shows how to use CSS class selectors to set the `labelPosition` and other properties that affect the appearance of data labels in a chart:

```
<?xml version="1.0"?>
<!-- charts/StylingDataLabels.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
      <!-- Style properties on series that affect data labels:
        fontFamily; fontSize; fontStyle; fontWeight;
        labelPosition; labelRotation; labelSizeLimit -->
       <fx:Style>
         .incomeSeries {
             fontSize:9;
             fontWeight:bold;
             labelPosition:inside;
             labelAlign:top;
         }

         .expensesSeries {
             fontSize:8;
             labelPosition:inside;
             labelAlign:middle;
         }

       </fx:Style>

    <s:layout>
```

```
        <s:VerticalLayout/>
    </s:layout>
       <s:Panel title="Column Chart">
             <s:layout>
                 <s:HorizontalLayout/>
             </s:layout>
          <mx:ColumnChart id="myChart"
             dataProvider="{srv.lastResult.data.result}"
             showDataTips="true">
             <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
             </mx:horizontalAxis>

             <mx:verticalAxis>
                 <mx:LinearAxis minimum="0" maximum="2500"/>
             </mx:verticalAxis>

             <mx:series>
                <mx:ColumnSeries
                     xField="month"
                     yField="profit"
                     displayName="Profit"
                     styleName="incomeSeries"/>
                <mx:ColumnSeries xField="month" yField="expenses"
                     displayName="Expenses" styleName="expensesSeries"/>
             </mx:series>
          </mx:ColumnChart>
          <mx:Legend dataProvider="{myChart}"/>
       </s:Panel>
</s:Application>
```

Labels are scaled and possibly truncated if they are too big for the area. Labels will never overlap each other or other chart elements.

For PieSeries objects, you can also specify the gap and stroke of callout lines that associate a data label with a particular wedge in the pie. For more information, see “Using data labels with PieChart controls” on page 1153.

Aligning data labels

You can align data labels so that they are positioned, either vertically or horizontally, relative to the underlying chart element. For example, you can center a data label over a column in a ColumnChart control, or align it to the left of all bars in a BarChart control.

To align data labels, you use the series’ `labelAlign` style property. For ColumnSeries objects, valid values are `middle`, `top`, and `bottom`. For BarSeries objects, valid values are `left`, `center`, and `right`. The defaults are `middle` and `center`, respectively.

You can only set the `labelAlign` style if the `labelPosition` property is set to `inside`. Label alignment is ignored if the `labelPosition` is `outside`.

You cannot change the alignment of data labels on a PieSeries object.

ColumnChart controls have two additional properties that you can use to control the way data labels are rendered: showLabelVertically and extendLabelToEnd. These properties are important when the available space for labels is not sufficient to render the entire label. If you set showLabelVertically to true, Flex can render the label vertically rather than horizontally if the columns are not wide enough to render them normally. If you set extendLabelToEnd to true, then Flex can use the space between the data item's edge and the outer boundary of the chart to render the label in, rather than truncate the label at the end of the data item's column.

Rotating data labels

You can rotate data labels on a series by setting the value of the labelRotation style property on the data label's axis renderer. You must embed a font to rotate them. You must also set the embedAsCFF property in CSS to false because the chart is an MX control and not a Spark control. If you rotate the data labels without embedding a font, they are rendered horizontally.

The following example embeds a font and rotates the data labels 45 degrees:

```
<?xml version="1.0"?>
<!-- charts/RotatingDataLabels.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Style>
        @font-face{
            src: url("../assets/MyriadWebPro.ttf");
            fontFamily: myMyriad;
            embedAsCFF: false;
        }
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|ColumnSeries {
            labelPosition:outside;
            labelRotation:45;
        }
        mx|ColumnChart {
            fontFamily: myMyriad;
        }
    </fx:Style>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Rotating Data Labels in a Column Chart">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
        <mx:ColumnChart id="myChart"
```

```
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">

            <mx:horizontalAxis>
                <mx:CategoryAxis id="a1" categoryField="month"/>
            </mx:horizontalAxis>
            <mx:verticalAxis>
                <mx:LinearAxis id="a2" minimum="0" maximum="2500"/>
            </mx:verticalAxis>
            <mx:series>
                <mx:ColumnSeries displayName="Profit"
                    xField="month"
                    yField="profit"/>
                <mx:ColumnSeries displayName="Expenses"
                    xField="month"
                    yField="expenses"/>
            </mx:series>
        </mx:ColumnChart>
    </s:Panel>
</s:Application>
```

You can only rotate data labels if the series's `labelPosition` property is set to `outside`. You cannot rotate data labels if the `labelPosition` property of the series is set to `inside`.

If by rotating labels, you cause them to overlap each other or overlap chart elements, Flex repositions them. If there is not enough space to reposition the labels, Flex scales and ultimately truncates the labels. Data labels will never overlap each other or the underlying chart elements.

For more information on embedding fonts, see "Embed fonts" on page 1571.

Customizing data label values

You can customize the value of your data labels. For example, you can prepend a dollar sign ($) or apply numeric formatters to the data labels to make your chart more readable. You can change the value of the data label altogther if you want.

To customize the contents of a data label, you use the `labelFunction` property of the series to specify a callback function. The function specified in `labelFunction` returns a String that Flex displays as the data label. Flex calls this callback function for each chart element in the series when the data labels are first rendered, and calls this method any time the labels are rendered again (for example, if the data changes).

The exact signature of the custom label callback function depends on the series that you are using it with. For BarSeries and ColumnSeries objects, the function takes two arguments (see "Customizing data labels for ColumnSeries and BarSeries objects" on page 1283); for PieSeries objects, the function takes four arguments (see "Customizing data labels for PieSeries objects" on page 1285).

Customizing data labels for ColumnSeries and BarSeries objects

For a ColumnSeries or BarSeries object, the signature for the custom label callback function is:

```
function_name(element:ChartItem, series:Series):String { ... }
```

The following table describes the parameters of the `labelFunction` callback function for a ColumnSeries or BarSeries object:

Parameter	Description
element	A reference to the ChartItem that this data label applies to. The ChartItem represents a single data point in a series.
series	A reference to the series that this data label is used on.

When you customize the value of the data label, you typically get a reference to the series item. You do this by casting the `element` argument to a specific chart item type (BarSeriesItem or ColumnSeriesItem). You then point to either the `yNumber` or `xNumber` property to get the underlying data.

You can also get a reference to the current series by casting the `series` argument to the specific series type (ColumnSeries or BarSeries). This lets you access properties such as the `yField` or `xField`.

The following example creates data labels for each of the columns in the ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/DataLabelFunctionColumn.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
         <s:NumberFormatter id="nf1" useGrouping="true" fractionalDigits="0"/>
    </fx:Declarations>
    <fx:Script><![CDATA[
        import mx.charts.ChartItem;
        import mx.charts.chartClasses.Series;
        import mx.charts.series.items.ColumnSeriesItem;
         private function setCustomLabel(element:ChartItem, series:Series):String {
            // Get a refereence to the current data element.
            var data:ColumnSeriesItem = ColumnSeriesItem(element);
            // Get a reference to the current series.
            var currentSeries:ColumnSeries = ColumnSeries(series);
            // Create a return String and format the number.
            return currentSeries.yField + ":" + " $" + nf1.format(data.yNumber);
         }
    ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        extendLabelToEnd="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis minimum="0" maximum="3000"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
                labelPosition="outside"
                xField="month"
                yField="profit"
                displayName="Profit"
                labelFunction="setCustomLabel"/>
           <mx:ColumnSeries
                labelPosition="outside"
                xField="month"
                yField="expenses"
                displayName="Expenses"
                labelFunction="setCustomLabel"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also access the specific field in the data provider that provides the data. For example, in the previous example, you could get the value of the data item (in this case, the value of the Income field) by using code similar to the following:

```
var item:ColumnSeriesItem = ColumnSeriesItem(element);
var s:String = item.item.Income;
return s;
```

This is not a recommended practice, however, because it makes the callback function less reusable and can force you to use additional code to detect which column you are providing data labels for.

Customizing data labels for PieSeries objects

For a PieSeries object, the signature for the custom label callback function is:

```
function_name(data:Object, field:String, index:Number, percentValue:Number):String { ... }
```

The following table describes the parameters of the `labelFunction` callback function for a PieSeries:

Parameter	Description
data	A reference to the data point that chart element represents; type Object.
field	The field name from the data provider; type String.
index	The number of the data point in the data provider; type Number.
percentValue	The size of the pie wedge relative to the pie; type Number. If the pie wedge is a quarter of the size of the pie, this value is 25.

The following example generates data labels for a PieSeries object that include data and formatting. It defines the `display()` method as the `labelFunction` callback function to handle formatting of the label text.

```
<?xml version="1.0"?>
<!-- charts/PieLabelFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import spark.formatters.*;
     public function display(
        data:Object,
        field:String,
        index:Number,
        percentValue:Number):String
     {
            return data.item + "\n$" + data.amount +
            "\n(" + round(percentValue,2) + "%)";
     }
     // Rounds to 2 places:
     public function round(num:Number, precision:Number):Number {
        var result:String;
        var f:NumberFormatter = new NumberFormatter();
        f.fractionalDigits = precision;
        result = f.format(num);
        return Number(result);
     }
  ]]></fx:Script>
    <s:layout>
```

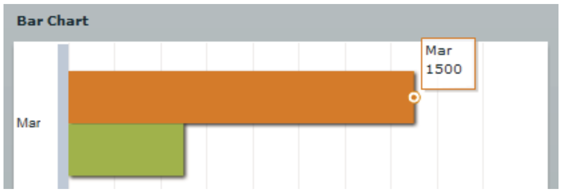
```
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Expenditures for FY '09">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PieChart id="chart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="false">
        <mx:series>
           <mx:PieSeries
                labelPosition="callout"
                field="amount"
                labelFunction="display"
                nameField="item"/>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{chart}"/>
  </s:Panel>
</s:Application>
```

Using DataTip objects

The values displayed in the DataTip depends on the chart type, but typically it displays the names of the fields and the values of the data from the data provider.

***Note:** DataTip controls and data labels are not the same, although they can show the same information. Data labels are always visible regardless of the location of the user's mouse pointer. For more information about data labels, see "Using data labels" on page 1277.*

The following image shows a simple DataTip:



To enable DataTip objects, set the value of the chart control's `showDataTips` property to `true`, as the following example shows:

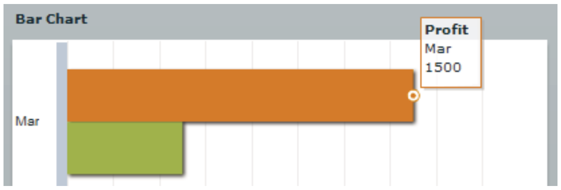
```
<?xml version="1.0"?>
<!-- charts/EnableDataTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart with DataTip objects enabled">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="Profit"/>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
      <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

To change the styles of the DataTip, you can use the DataTip type selector in an `<fx:Style>` block or in an external CSS file. You must create a separate style namespace definition for the DataTip package. The following example applies new style properties to the text in the DataTip:

```
<?xml version="1.0"?>
<!-- charts/DataTipStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
          <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
          <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Style>
        @namespace chartClasses "mx.charts.chartClasses.*";
        chartClasses|DataTip {
            fontFamily: "Arial";
            fontSize: 12;
            fontWeight:bold;
            fontStyle:italic;
        }
    </fx:Style>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Bar Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:BarChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:verticalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:verticalAxis>
            <mx:series>
                <mx:BarSeries
                    yField="month"
                    xField="profit"
                    displayName="Profit"/>
                <mx:BarSeries
                    yField="month"
                    xField="expenses"
                    displayName="Expenses"/>
            </mx:series>
        </mx:BarChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

Customizing the text inside a DataTip object

To make the information in the DataTip more understandable to users, you can define the series of your chart with names that are easily understood. Adobe® Flash® Player or Adobe® AIR™ displays this name in the DataTip, as the following image shows:

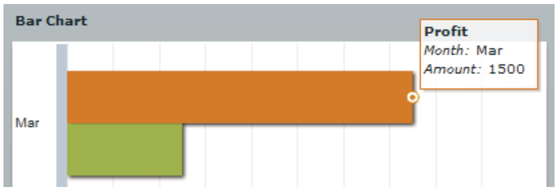


The following example names the data series by using the `displayName` property of the series:

```
<?xml version="1.0"?>
<!-- charts/DataTipsDisplayName.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
```

```
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="--==Profit==--"/>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="--==Expenses==--"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also name the axis to display labels in the DataTip by using the `displayName` property. When the axis has a name, this name appears in the DataTip in italic font before the label data, as the following image shows:



In some cases, you add an axis solely for the purpose of adding the label to the DataTip. The following example names both axes so that both data points are labeled in the DataTip:

```
<?xml version="1.0"?>
<!-- charts/DataTipsAxisNames.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <s:Panel title="Bar Chart">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
         <mx:BarChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:verticalAxis>
               <mx:CategoryAxis categoryField="month"
                    displayName="Month"/>
            </mx:verticalAxis>
            <mx:horizontalAxis>
               <mx:LinearAxis displayName="Amount"/>
            </mx:horizontalAxis>
            <mx:series>
               <mx:BarSeries
                    yField="month"
                    xField="profit"
                    displayName="Profit"/>
               <mx:BarSeries
                    yField="month"
                    xField="expenses"
                    displayName="Expenses"/>
            </mx:series>
         </mx:BarChart>
         <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

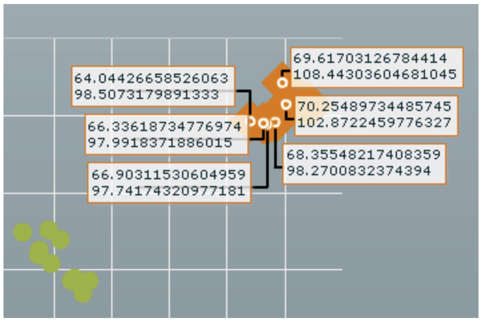
Showing multiple DataTip objects

You can display more than one DataTip by using the `dataTipMode` property on the chart control. The display options are `single` and `multiple`. When `dataTipMode` is set to `multiple`, the chart displays all DataTip objects within range of the cursor. The following example sets the value of a ColumnChart control's `dataTipMode` property to `multiple`:

```
<?xml version="1.0"?>
<!-- charts/DataTipsMultiple.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
```

```
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        mouseSensitivity="50"
        dataTipMode="multiple">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="Profit"/>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The following example shows DataTip objects when the `dataTipMode` property is set to `multiple`:



The default value of `dataTipMode` depends on the chart type. Its setting is based on the likelihood that there are overlapping DataTip objects in that chart type. The default value of the `dataTipMode` property for the following chart types is `single`:

- BarChart
- CandlestickChart
- ColumnChart
- HLOCChart
- PieChart

The default value is `multiple` for the `dataTipMode` property for all other chart types.

To determine the size of the interactive area around a data point, you set the `mouseSensitivity` property. The `mouseSensitivity` property configures the distance, in pixels, around the data points where Flex reacts to mouse events such as `click` and `mouseOver`. With this property, you can trigger DataTip objects to appear when the user moves the mouse pointer *near* the data point rather than *onto* the data point. For more information, see “Changing mouse sensitivity” on page 1335.

You can also show all the available data tips on a chart at one time by setting the value of the chart control's `showAllDataTips` property to `true`. The result is that all data tips are visible at all times. When you do this, you typically set the value of the `showDataTips` property to `false` so that a second data tip does not appear when you mouse over a chart item.

Customizing DataTip values

You can customize the text displayed in a DataTip by using the `dataTipFunction` callback function. When you specify a `dataTipFunction` callback function, you can access the data of the DataTip before Flex renders it and customizes the text.

The argument to the callback function is a HitData object. As a result, you must import mx.charts.HitData when using a DataTip callback function.

Flex displays whatever the callback function returns in the DataTip box. You must specify a String as the callback function's return type.

The following example defines a new callback function, `dtFunc`, that returns a formatted value for the DataTip:

```
<?xml version="1.0"?>
<!-- charts/CustomDataTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.charts.HitData;
     public function dtFunc(hd:HitData):String {
        return hd.item.month + ": " +
        "<B>$" + hd.item.profit + "</B>";
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        dataTipFunction="dtFunc">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month" displayName="Month"/>
        </mx:verticalAxis>
        <mx:horizontalAxis>
           <mx:LinearAxis displayName="Amount"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="Profit"/>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You can also use the HitData object to get information about the series in which that data item appears. To do this, you cast the HitData object to a Series class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HitDataCasting.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.charts.HitData;
     import mx.charts.series.ColumnSeries;
     public var b:Boolean = true;
     public function myDataTipFunction(e:HitData):String {
        var s:String;
        s = ColumnSeries(e.element).displayName + "\n";
        s += "Profit: $" + (e.item.profit - e.item.expenses);
        return s;
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Casting HitData Objects">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        dataTipFunction="myDataTipFunction">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="profit"
                displayName="Income '09"/>
           <mx:ColumnSeries
                yField="expenses"
                displayName="Expenses '09"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The series item is also accessible from the HitData object in a custom DataTip function. The `chartItem` property refers to an instance of a subclass of the ChartItem class. The type depends on the series type; for example, the `chartItem` for a ColumnSeries is an instance of the ColumnSeriesItem class.

In the following example, the `yValue` of the ColumnSeriesItem represents the percentage which a series takes up in a 100% chart:

```
<?xml version="1.0"?>
<!-- charts/HitDataCastingWithPercent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.charts.HitData;
     import mx.charts.series.ColumnSeries;
     import mx.charts.series.items.ColumnSeriesItem;
     public var b:Boolean = true;
     public function myDataTipFunction(e:HitData):String {
        var s:String;
        s = "<B>" + ColumnSeries(e.element).displayName + "</B>\n";
        s += "<I>Profit:</I> <FONT COLOR='#339966'>$" +
            e.item.profit + "</FONT>\n";
        s += "<I>Expenses:</I> <FONT COLOR='#FF0000'>$" +
            e.item.expenses + "</FONT>\n";
        s += "------------------------\n";
        s += "<I>Difference:</I> $" + (e.item.profit - e.item.expenses) + "\n";
        // The value of the Profit will always be 100%,
        // so exclude adding that to the DataTip. Only
        // add percent when the user gets the Amount DataTip.
        var percentValue:Number = Number(ColumnSeriesItem(e.chartItem).yValue);
        if (percentValue < 100) {
            s += "Profit was equal to about <B>" +
                Math.round(percentValue) + "</B>% of the total.\n";
        }
        return s;
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Accessing ChartItems from HitData Objects">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        type="100%"
        dataTipFunction="myDataTipFunction"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="profit"
                displayName="Profit '09"/>
           <mx:ColumnSeries
                yField="expenses"
                displayName="Expenses '09"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

For more information on using the HitData object of chart events, see "Using the HitData object" on page 1327.

The DataTip callback function can render simple HTML for formatted DataTip objects. Flex supports a small subset of HTML tags including `<FONT>`, `<B>`, `<I>`, and `<BR>`.

Creating custom DataTip renderers

You can create a custom class that defines the appearance of the data tip. This class must implement the IFlexDisplayObject and IDataRenderer interfaces.

You typically create a custom ActionScript class that extends DataTip, and override the `updateDisplayList()` method. In the `updateDisplayList()` method, you define the appearance of the box for the data tip and apply styles that apply to the text for the data tip.

The following example class creates a custom data tip:

```
// charts/MyDataTip.as
package {
    import mx.charts.chartClasses.DataTip;
    import mx.charts.*;
    import flash.display.*;
    import flash.geom.Matrix;

    public class MyDataTip extends DataTip {
        public function MyDataTip() {
            super();
        }

        override protected function updateDisplayList(w:Number, h:Number):void {
            super.updateDisplayList(w, h);

            this.setStyle("textAlign","center");
            var g:Graphics = graphics;
            g.clear();
            var m:Matrix = new Matrix();
            m.createGradientBox(w+100,h,0,0,0);
            g.beginGradientFill(GradientType.LINEAR,[0xFF0000,0xFFFFFF],
                [.1,1],[0,255],m,null,null,0);
            g.drawRect(-50,0,w+100,h);
            g.endFill();
        }
    }
}
```

To use the custom data tip, you set the value of the `dataTipRenderer` style property of the chart control to the custom class. The following sample application applies the custom data tip to the chart control.

```
<?xml version="1.0"?>
<!-- charts/CustomDataTipRenderer.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();applyCustomDataTips();"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
          <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
          <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
    private function applyCustomDataTips():void {
        myChart.setStyle("dataTipRenderer",MyDataTip);
    }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="month"
                xField="profit"
                displayName="Profit"/>
           <mx:BarSeries
                yField="month"
                xField="expenses"
                displayName="Expenses"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

If you want to prevent the data tip's box from displaying when you mouse over a chart item, you can set the `dataTipRenderer` style property to a dummy class, such as ProgrammaticSkin; for example:

```
myChart.setStyle("dataTipRenderer",mx.skins.ProgrammaticSkin);
```

If you want to change the text of the data tip, you use the `dataTipFunction` callback function on the chart control, as described in "Customizing DataTip values" on page 1294.

Using per-item fills

You can customize the appearance of chart items in a series by using the `fillFunction` property to define the fill. This function takes the chart item and its index as arguments, so that you can examine the chart item's data values and return a fill based on whatever criteria you choose to use.

Programmatically assigning fills lets you set a threshold for color values, or conditionalize the colors of your chart items. For example, if the size of a pie wedge is greater than 25%, make it red, or if a column's value is greater than 100, make it green.

You set the value of the `fillFunction` property on each series, so if you have multiple series, you can have multiple fill functions, or all series can share the same fill function.

The signature of the `fillFunction` is as follows:

```
function_name(element:ChartItem, index:Number):IFill { ... }
```

The following table describes the arguments:

Argument	Description
element	The chart item for which the fill is created; type ChartItem.
index	The index of the chart item in the RenderData object's cache. This is different from the index of the chart's data provider because it is sorted based on the x, y, and z values; type Number.

The fillFunction property returns a Fill object (an object that implements the IFill interface). This object is typically an instance of the SolidColor class, but it can also be of type BitmapFill, LinearGradient, or RadialGradient. For information on working with gradients, see “Using gradient fills with chart controls” on page 1206.

The returned fill from a fillFunction takes precedence over fills that are set with traditional style methods. For example, if you set the value of the fill or fills property of a series and also specify a fillFunction, the fills are ignored and the fill returned by the fillFunction is used when rendering the chart items in the series.

The following example compares the value of the yField in the data provider when it fills each chart item. If the yField value (corresponding to the CurrentAmount field) is greater than $50,000, the column is green. If it is less than $50,000, the column is red.

```
<?xml version="1.0"?>
<!-- charts/SimpleFillFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script>
    <![CDATA[
     import mx.graphics.IFill;
     import mx.graphics.SolidColor;
     import mx.charts.ChartItem;
     import mx.charts.series.items.ColumnSeriesItem;
    private function myFillFunction(element:ChartItem, index:Number):IFill {
        var c:SolidColor = new SolidColor(0x00CC00);
        var item:ColumnSeriesItem = ColumnSeriesItem(element);
        var profit:Number = Number(item.yValue);
        if (profit >= 1250) {
            return c;
        } else {
            c.color = 0xFF0000;
        }
        return c;
    }
    ]]>
  </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Using a custom fillFunction in a Column Chart">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis title="Month" categoryField="month"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis title="Profit (in $USD)"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="currSalesSeries"
                xField="month"
                yField="profit"
                fillFunction="myFillFunction"
                displayName="Profit"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend>
        <mx:LegendItem label="More than $1,250" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0x00FF00"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0x000000" weight="1"/>
           </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="Less than $1,250" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0xFF0000"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0x000000" weight="1"/>
           </mx:stroke>
         </mx:LegendItem>
     </mx:Legend>
  </s:Panel>
</s:Application>
```

This example defines custom entries in the Legend. If you use the `fillFunction` property to define the fills of chart items, and you want a Legend control in your chart, you must manually create the Legend object and its LegendItem objects. For more information on creating Legend and LegendItem objects, see “Using Legend controls” on page 1316.

The color returned by the fill function is used by the itemRenderer for the series. In the case of the ColumnSeries and BarSeries classes, this is the column or bar in the chart. For LineSeries and AreaSeries classes, however, the renderer is used on the line itself. As a result, the color is applied only if you define a custom itemRenderer (such as a CircleItemRenderer or DiamondItemRenderer), and then it is applied only to the that renderer (such as the color inside the diamond) along the line of those series, not to the fill area below the line as you might expect.

By using the `index` argument that is passed to the fill function, you can also access other items inside the same series.

The following example sets the color of the fill to green for only the columns whose item value is greater than the previous item’s value. Otherwise, it sets the color of the fill to red.

```
<?xml version="1.0"?>
<!-- charts/ComparativeFillFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script>
    <![CDATA[
     import mx.graphics.IFill;
     import mx.graphics.SolidColor;
     import mx.charts.ChartItem;
     import mx.charts.series.items.ColumnSeriesItem;
    private function myFillFunction(element:ChartItem, index:Number):IFill {
        // Default to green.
        var c:SolidColor = new SolidColor(0x00FF00);
        var item:ColumnSeriesItem = ColumnSeriesItem(element);
        var profit:Number = Number(item.yValue);
        if (index == 0) {
            // The first column should be green, no matter the value.
            return c;
        } else {
            var prevVal:Number = 
                Number(currSalesSeries.items[index - 1].yValue);
            var curVal:Number = 
                Number(currSalesSeries.items[index].yValue);
            var diff:Number = curVal - prevVal;
            if (diff >= 0) {
                // Current column's value is greater than the previous.
                return c;
            } else {
                // Previous column's value is greater than the current.
                c.color = 0xFF0000;
            }
        }
        return c;
    }
    ]]>
  </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Using a custom fillFunction in a Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                title="Month"
                categoryField="month"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis title="Profit (in $USD)"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="currSalesSeries"
                xField="month"
                yField="profit"
                fillFunction="myFillFunction"
                displayName="Profit"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend>
        <mx:LegendItem label="More than Previous" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0x00FF00"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0x000000" weight="1"/>
           </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="Less than Previous" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0xFF0000"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0x000000" weight="1"/>
           </mx:stroke>
         </mx:LegendItem>
     </mx:Legend>
  </s:Panel>
</s:Application>
```

This example defines custom entries in the Legend because using a `fillFunction` prevents you from using an automatically generated Legend. For more information on creating Legend objects, see "Using Legend controls" on page 1316.

You can compare the values of data in a chart item to other values in the same or different series. The following example compares the value of the current amount of sales in the currentSalesSeries series against the sales goal in the salesGoalSeries series. If the goal is met or exceeded, the column is green. If the goal is not yet met, the negative difference between the goal and the actual sales is red.

```
<?xml version="1.0"?>
<!-- charts/ComplexFillFunction.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
  <fx:Script>
    <![CDATA[
     import mx.controls.Alert;
     import mx.graphics.IFill;
     import mx.graphics.SolidColor;
     import mx.collections.ArrayCollection;
     import mx.charts.ChartItem;
     import mx.charts.series.items.ColumnSeriesItem;
     [Bindable]
     public var sales:ArrayCollection = new ArrayCollection([
        { Name:"Reiner", SalesGoal:65000, CurrentAmount:69000 },
        { Name:"Alan", SalesGoal:40000, CurrentAmount:44000 },
        { Name:"Wolfgang", SalesGoal:48000, CurrentAmount:33000 },
        { Name:"Francis", SalesGoal:22000, CurrentAmount:20000 },
        { Name:"Klaus", SalesGoal:50000, CurrentAmount:55000 },
        { Name:"Martin", SalesGoal:44000, CurrentAmount:70000 },
        { Name:"Mac", SalesGoal:40000, CurrentAmount:35000 },
        { Name:"Friedemann", SalesGoal:38000, CurrentAmount:38000 },
        { Name:"Bruno", SalesGoal:42000, CurrentAmount:40000 }
     ]);
    private function myFillFunction(element:ChartItem, index:Number):IFill {
        var item:ColumnSeriesItem = ColumnSeriesItem(element);
        // Set default color.
        var c:SolidColor = new SolidColor(0xFF3366, 1);

        /* Use the yNumber properties rather than the yValue properties
           because the conversion to a Number is already done for
           you. As a result, you do not have to cast them to a Number,
           which would be less efficient. */

        var currentAmount:Number = currSalesSeries.items[index].yNumber;
        var goal:Number = item.yNumber;
        // Determine if the goal was met or not.
        var diff:Number = currentAmount - goal;

        if (diff >= 0) {
            // Sales person met their goal.
            return c;
        } else if (diff < 0) {
            // Sales person did not meet their goal.
            c.color = 0xFF0000;
            c.alpha = 1;
        }
        return c;
    }

    private function showDataPoints():void {
        var s:String = "";
        for (var i:int = 0; i<currSalesSeries.items.length; i++) {
            if (salesGoalSeries.items[i].yNumber != null && currSalesSeries.items[i].yNumber
!= null) {
                var currentAmount:Number = currSalesSeries.items[i].yNumber;
                var goal:Number = salesGoalSeries.items[i].yNumber;
                var diff:Number = currentAmount - goal;
                s += "c:" + currentAmount.toString() + ", g:" + goal.toString() + ", d:" +
diff.toString() + "\n";
            }
        }
        Alert.show(s);
    }
    ]]>
  </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Using a custom fillFunction in a Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{sales}"
        type="overlaid"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                title="Sales Person"
                categoryField="Name"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis title="Sales (in $USD)"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="salesGoalSeries"
                xField="Name"
                yField="SalesGoal"
                fillFunction="myFillFunction"
                displayName="Sales Goal">
           </mx:ColumnSeries>
           <mx:ColumnSeries id="currSalesSeries"
                xField="Name"
                yField="CurrentAmount"
                displayName="Current Sales">
                <mx:fill>
                    <mx:SolidColor color="0x00FF00"/>
                </mx:fill>
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend>
```

```
        <mx:LegendItem label="Current Sales" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0x00FF00"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0x000000" weight="1"/>
           </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="Missed Goal" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0xFF0000"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0x000000" weight="1"/>
           </mx:stroke>
         </mx:LegendItem>
     </mx:Legend>
     <s:Button click="showDataPoints()" label="Show Data"/>
  </s:Panel>
</s:Application>
```

You can access data from other series inside a series's fill function, but only if the other series has been rendered. If you were to rearrange the order of the series in the previous example, Flash would throw an error because the second series has not yet been rendered when the fill function is called.

Using the minField property

Some series types let you specify a minimum value for the elements drawn on the screen. For example, in a ColumnChart control, you can specify the base value of the column. To specify a base value (or minimum) for the column, set the value of the series object's `minField` property to the data provider field.

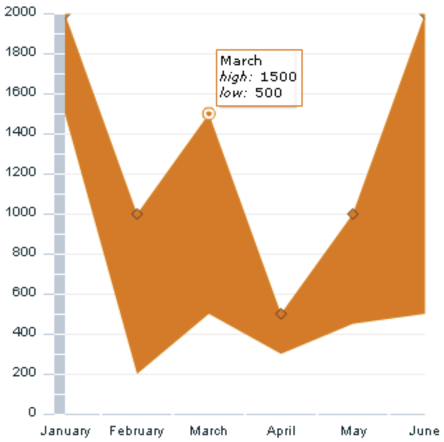
You can specify the `minField` property for the following chart series types:

- AreaSeries
- BarSeries
- ColumnSeries

Setting a value for the `minField` property creates two values on the axis for each data point in an area; for example:

```
<?xml version="1.0"?>
<!-- charts/MinField.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Area Chart">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
     <mx:AreaChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries 
                yField="profit" 
                minField="expenses" 
                displayName="Profit"/>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The resulting DataTip labels the current value “high” and the `minField` value “low.” The following example shows an AreaChart that defines the base of each column:



For an example of using the `minField` property to create a waterfall or cascading ColumnChart control, see “Using column charts” on page 1137.

You can also use the `baseAtZero` property to determine whether or not the axis starts at zero. Set this property to `true` to start the axis at zero; set it to `false` to let Flex determine a reasonable starting value relative to the values along the axis.

Stacking charts

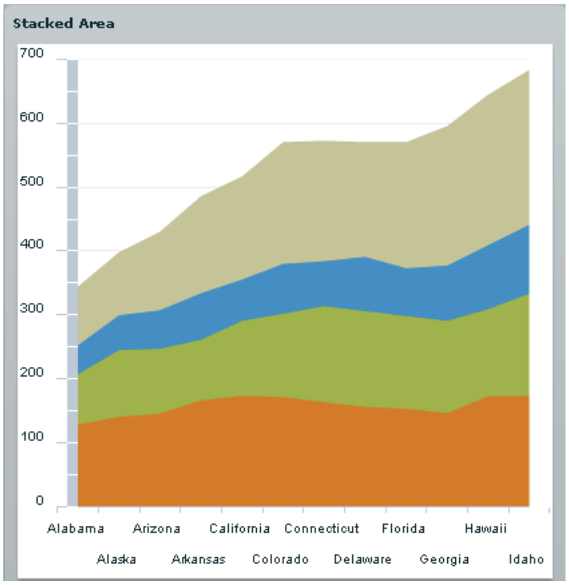
When you use multiple data series in the AreaChart, BarChart, and ColumnChart controls, you can control how Flex displays the series using the `type` property of the controls. The following table describes the values that the `type` property supports:

Property	Description
`clustered`	Chart elements for each series are grouped by category. This is the default value for BarChart and ColumnChart controls.
`overlaid`	Chart elements for each series are rendered on top of each other, with the element corresponding to the last series on top. This is the default value for AreaChart controls.
`stacked`	Chart elements for each series are stacked on top of each other. Each element represents the cumulative value of the elements beneath it.
`100%`	Chart elements are stacked on top of each other, adding up to 100%. Each chart element represents the percentage that the value contributes to the sum of the values for that category.

The following example creates an AreaChart control that has four data series, stacked on top of each other:

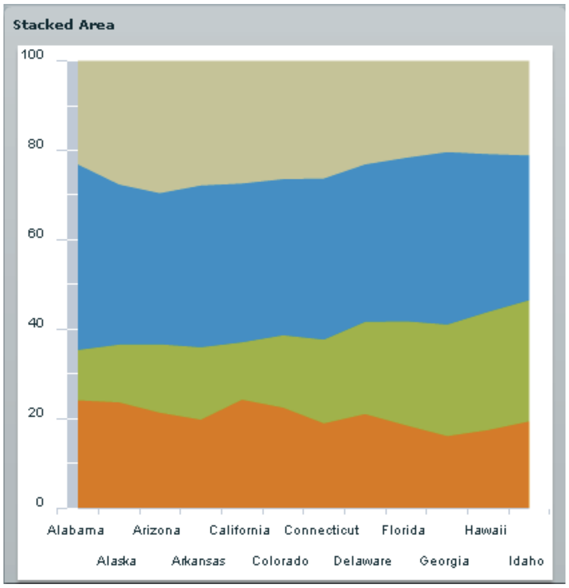
```
<?xml version="1.0"?>
<!-- charts/AreaStacked.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Stacked AreaChart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:AreaChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        type="stacked">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries yField="profit" displayName="Profit"/>
           <mx:AreaSeries yField="expenses" displayName="Expenses"/>
           <mx:AreaSeries yField="amount" displayName="Amount"/>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The following example shows a stacked AreaChart control:



With an overlay, the last series appears on top, and can obscure the data series below it unless you use the `alpha` property of the fill to make it transparent. For more information, see "Using fills with chart controls" on page 1197.

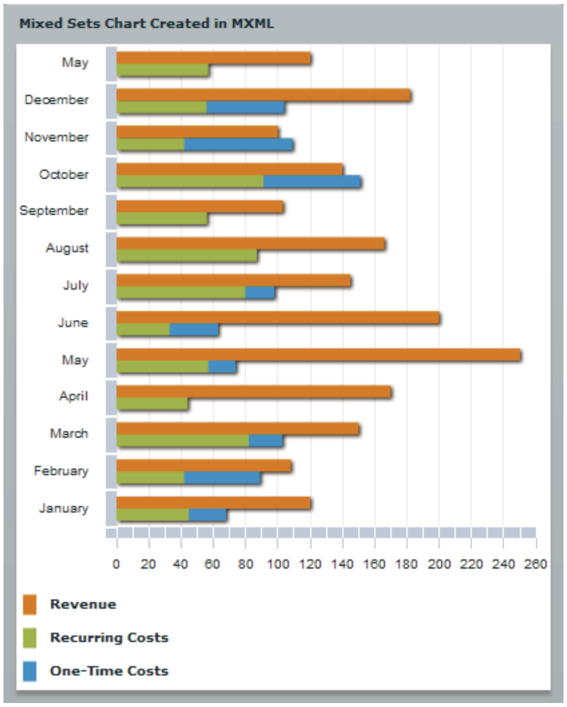
If you set the `type` property to 100%, the control draws each series stacked on top of each other, adding up to 100% of the area. Each column represents the percentage that the value contributes to the sum of the values for that category, as the following example shows:



You can use the ColumnSet, BarSet, and AreaSet classes to combine groups of chart series, and thereby use different types of series within the same chart. The following example uses BarSet classes to combine clustered and stacked BarSeries in a single chart. The example shows how to do this in MXML and ActionScript:

```
<?xml version="1.0"?>
<!-- charts/UsingBarSets.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()"
    height="1050" width="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
     import mx.charts.Legend;
     import mx.charts.BarChart;
     import mx.charts.series.BarSet;
     import mx.charts.series.BarSeries;
     import mx.collections.ArrayCollection;
     [Bindable]
     private var yearlyData:ArrayCollection = new ArrayCollection([
        {month:"January", revenue:120, costs:45,
            overhead:102, oneTime:23},
        {month:"February", revenue:108, costs:42,
            overhead:87, oneTime:47},
        {month:"March", revenue:150, costs:82,
            overhead:32, oneTime:21},
        {month:"April", revenue:170, costs:44,
            overhead:68},
        {month:"May", revenue:250, costs:57,
            overhead:77, oneTime:17},
        {month:"June", revenue:200, costs:33,
            overhead:51, oneTime:30},
        {month:"July", revenue:145, costs:80,
            overhead:62, oneTime:18},
        {month:"August", revenue:166, costs:87,
            overhead:48},
        {month:"September", revenue:103, costs:56,
            overhead:42},
        {month:"October", revenue:140, costs:91,
            overhead:45, oneTime:60},
        {month:"November", revenue:100, costs:42,
            overhead:33, oneTime:67},
        {month:"December", revenue:182, costs:56,
            overhead:25, oneTime:48},
        {month:"May", revenue:120, costs:57,
            overhead:30}
     ]);
     private function initApp():void {
        var c:BarChart = new BarChart();
        c.dataProvider = yearlyData;
        c.showDataTips = true;
        var vAxis:CategoryAxis = new CategoryAxis();
        vAxis.categoryField = "month";
        c.verticalAxis = vAxis;
        var mySeries:Array = new Array();
        var outerSet:BarSet = new BarSet();
        outerSet.type = "clustered";
        var series1:BarSeries = new BarSeries();
        series1.xField = "revenue";
        series1.displayName = "Revenue";
        outerSet.series = [series1];
        var innerSet:BarSet = new BarSet();
        innerSet.type = "stacked";
        var series2:BarSeries = new BarSeries();
        var series3:BarSeries = new BarSeries();
        series2.xField = "costs";
        series2.displayName = "Recurring Costs";
        series3.xField = "oneTime";
        series3.displayName = "One-Time Costs";
        innerSet.series = [series2, series3];
        c.series = [outerSet, innerSet];
        var l:Legend = new Legend();
        l.dataProvider = c;
        panel2.addElement(c);
        panel2.addElement(l);
     }
  ]]></fx:Script>
  <s:Panel title="Mixed Sets Chart Created in MXML" id="panel1">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{yearlyData}" showDataTips="true">
        <mx:verticalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
            <mx:BarSet type="clustered">
                <mx:BarSeries xField="revenue"
                    displayName="Revenue"/>
                <mx:BarSet type="stacked">
                    <mx:BarSeries
                        xField="costs"
                        displayName="Recurring Costs"/>
                    <mx:BarSeries
                        xField="oneTime"
                        displayName="One-Time Costs"/>
                </mx:BarSet>
            </mx:BarSet>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
  <s:Panel title="Mixed Sets Chart Created in ActionScript" id="panel2">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  </s:Panel>
</s:Application>
```

The resulting chart shows two clustered series; one is a standalone series, and the other is a stacked series, as the following example shows:



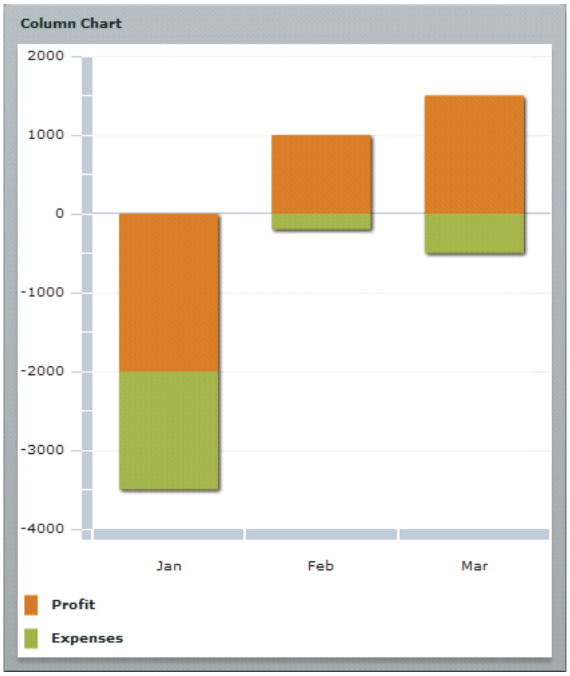
Normally, the values in stacked series are additive, which means that they are all added to one another to create an ever larger data item (column or bar or area). When one or more values is negative, the rendering of the data items can be unpredictable because adding a negative value would normally take away from the data item. You can also use the `allowNegativeForStacked` property of the AreaSet, BarSet, and ColumnSet classes to let a stacked series include data with negative values, and thereby render negative data items that are properly stacked.

The following example stacks the Profit and Expenses fields, in which some of the values are negative.

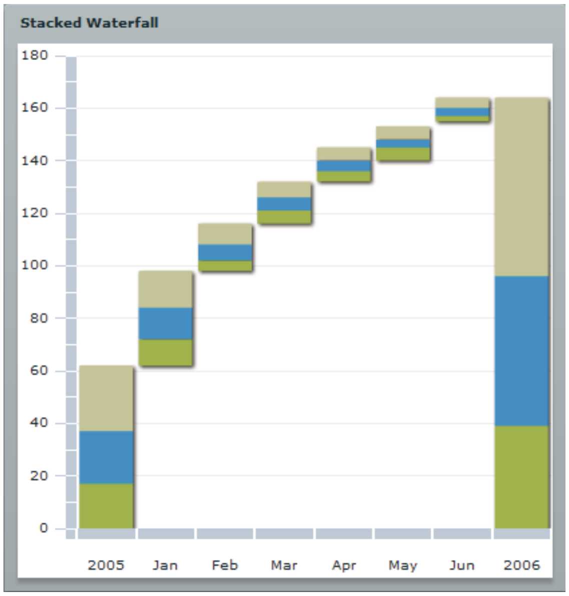
```
<?xml version="1.0"?>
<!-- charts/StackedNegative.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:-2000, Expenses:-1500},
        {Month:"Feb", Profit:1000, Expenses:-200},
        {Month:"Mar", Profit:1500, Expenses:-500}
     ]);
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
```

```
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSet type="stacked"
                allowNegativeForStacked="true">
                <mx:series>
                    <mx:ColumnSeries
                        xField="Month"
                        yField="Profit"
                        displayName="Profit"/>
                    <mx:ColumnSeries
                        xField="Month"
                        yField="Expenses"
                        displayName="Expenses"/>
                </mx:series>
            </mx:ColumnSet>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The resulting chart shows three months of data, with all expenses and some income data rendering in negative values:



You can create cascading or waterfall column charts by using the ColumnChart control. One way to do this is to create an invisible series and to use that to set the variable height of the other columns, creating the waterfall effect. The following is an example of a waterfall chart:

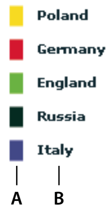


The following code creates this chart:

```
<?xml version="1.0"?>
<!-- charts/WaterfallStacked.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">

  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"2005", top:25, middle:20, bottom:17, Invisible:0},
        {Month:"Jan", top:14, middle:12, bottom:10, Invisible:62},
        {Month:"Feb", top:8, middle:6, bottom:4, Invisible:98},
        {Month:"Mar", top:6, middle:5, bottom:5, Invisible:116},
        {Month:"Apr", top:5, middle:4, bottom:4, Invisible:132},
        {Month:"May", top:5, middle:3, bottom:5, Invisible:140},
        {Month:"Jun", top:4, middle:3, bottom:2, Invisible:155},
        {Month:"2006", top:68, middle:57, bottom:39, Invisible:0}
     ]);
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Stacked Waterfall">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        columnWidthRatio=".9"
        showDataTips="true"
        type="stacked">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries yField="Invisible">
                <mx:fill>
                    <!--Set alpha to 0 to hide invisible column.-->
                    <mx:SolidColor color="0xFFFFFF" alpha="0"/>
                </mx:fill>
           </mx:ColumnSeries>
           <mx:ColumnSeries yField="bottom" displayName="Profit"/>
           <mx:ColumnSeries yField="middle" displayName="Expenses"/>
           <mx:ColumnSeries yField="top" displayName="Profit"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

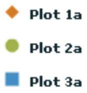
Using Legend controls

Legend controls match the fill patterns on your chart to labels that describe the data series shown with those fill patterns. Each entry in a Legend control is known as a legend item. A legend item contains two basic parts: the marker, and the label. Legend items are of type LegendItem. The following example shows the two parts of a legend item:



***A.** Marker **B.** Label*

In addition to matching fill patterns, legend markers also use the renderer classes of ChartItem objects, such as the points on a PlotChart control. The following example shows a Legend control for a PlotChart control with three series. Each series uses its own renderer class to draw a particular shape, and the Legend control reflects those shapes:



For information on styling Legend controls, see “Formatting Legend controls” on page 1241.

Scrollbars are not supported on Legend controls. If you want to add scrollbars, wrap your Legend control in a container, and add scrollbars to the container.

Adding a Legend control to your chart

You use the Legend class to add a legend to your charts. The Legend control displays the label for each data series in the chart and a key that shows the chart element for the series.

The simplest way to create a Legend control is to bind a chart to it by using the `dataProvider` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendNamedSeries.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Bar Chart with Legend">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="amount"
                displayName="Amount (in $USD)"/>
           <mx:BarSeries
                xField="profit"
                displayName="Profit (in $USD)"/>
           <mx:BarSeries
                xField="expenses"
                displayName="Expense (in $USD)"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

You add a Legend to a chart in ActionScript by instantiating a new object of type Legend, and then calling the container's `addElement()` method to add the Legend to the container, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendInAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx" result="createLegend()"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.charts.Legend;
     private function createLegend():void {
        var myLegend:Legend = new Legend();
        myLegend.dataProvider = myChart;
        panel1.addElement(myLegend);
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel id="panel1" title="Bar Chart with Legend (Created in ActionScript)">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="amount"
                displayName="Amount (in $USD)"/>
           <mx:BarSeries
                xField="profit"
                displayName="Profit (in $USD)"/>
           <mx:BarSeries
                xField="expenses"
                displayName="Expense (in $USD)"/>
        </mx:series>
     </mx:BarChart>
  </s:Panel>
</s:Application>
```

The Legend control creates the legend using information from the chart control. It matches the colors and names of the LegendItem markers to the fills and labels of the chart's data series. In the previous example, Flex uses the BarSeries control's `displayName` property to define the LegendItem label.

Legend controls require that elements of the charts be named. If they are not named, the Legend markers appear, but without labels.

A Legend control for a PieChart control uses the `nameField` property of the data series to find values for the legend. The values that the `nameField` property point to must be Strings.

The following example sets the `nameField` property of a PieChart control's data series to `item`. Flex uses the value of the `item` field in the data provider in the Legend control.

```
<?xml version="1.0"?>
<!-- charts/LegendNameField.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/budget-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Pie Chart with Legend">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PieChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true">
        <mx:series>
           <mx:PieSeries field="amount" nameField="item"/>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

The `nameField` property also defines the series for the DataTip objects and labels.

Creating a custom Legend control

You can create a custom Legend control in MXML by defining the `<mx:Legend>` tag and populating it with `<mx:LegendItem>` tags. The following example creates a custom legend for the chart with multiple axes:

```
<?xml version="1.0"?>
<!-- charts/MultipleAxesWithCustomLegend.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_fe.send();srv_strk.send();"
    height="600">
    <fx:Declarations>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
       <!-- View source of the following pages to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv_fe" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FE"/>
       <mx:HTTPService id="srv_strk" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=STRK"/>
          <mx:SolidColorStroke id="h1Stroke"
                color="{feColor}"
                weight="8" alpha=".75"
                caps="square"/>
          <mx:SolidColorStroke id="h2Stroke"
                color="{strkColor}"
                weight="8" alpha=".75"
                caps="square"/>
    </fx:Declarations>
  <fx:Script><![CDATA[
    [Bindable]
    public var strkColor:Number = 0x224488;
    [Bindable]
    public var feColor:Number = 0x884422;
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart With Multiple Axes">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart" showDataTips="true">
        <mx:horizontalAxis>
           <mx:DateTimeAxis id="h1" dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer placement="bottom" axis="{h1}"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
            <mx:AxisRenderer placement="left" axis="{v1}">
                <mx:axisStroke>{h1Stroke}</mx:axisStroke>
            </mx:AxisRenderer>
            <mx:AxisRenderer placement="left" axis="{v2}">
                <mx:axisStroke>{h2Stroke}</mx:axisStroke>
            </mx:AxisRenderer>
        </mx:verticalAxisRenderers>
        <mx:series>
           <mx:ColumnSeries id="cs1"
                horizontalAxis="{h1}"
                dataProvider="{srv_fe.lastResult.data.result}"
                yField="close"
                xField="date"
                displayName="FE">
                <mx:fill>
                    <mx:SolidColor color="{feColor}"/>
                </mx:fill>
                <mx:verticalAxis>
                   <mx:LinearAxis id="v1"/>
                </mx:verticalAxis>
           </mx:ColumnSeries>
           <mx:LineSeries id="cs2"
                horizontalAxis="{h1}"
                dataProvider="{srv_strk.lastResult.data.result}"
                yField="close"
                xField="date"
                displayName="STRK">
                <mx:verticalAxis>
                    <mx:LinearAxis id="v2"/>
                </mx:verticalAxis>
                <mx:lineStroke>
                    <mx:SolidColorStroke
                        color="{strkColor}"
                        weight="4"
                        alpha="1"/>
                </mx:lineStroke>
           </mx:LineSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend>
        <mx:LegendItem label="FE" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="{feColor}"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0xCCCCCC" weight="2"/>
           </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="STRK" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="{strkColor}"/>
           </mx:fill>
           <mx:stroke>
            <mx:SolidColorStroke color="0xCCCCCC" weight="2"/>
           </mx:stroke>
         </mx:LegendItem>
     </mx:Legend>
  </s:Panel>
</s:Application>
```

To create a custom Legend control in ActionScript, you create a container and add LegendItem objects to it. You then apply the appropriate styles and properties to the LegendItem objects. You can get each LegendItem's label and fill color from the corresponding series in the chart's series array.

The following example creates a custom legend in ActionScript. It lays the LegendItem objects out in a grid, with GridRows and GridItems. This example lets the user customize the number of LegendItem objects in each GridRow. You can use any layout container you want to get the desired appearance of your custom legend.

```
<?xml version="1.0"?>
<!-- charts/CustomLegendInActionScript.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="drawLegend()"
    height="600">

  <fx:Script><![CDATA[
     import mx.containers.GridItem;
     import mx.containers.GridRow;
     import mx.graphics.SolidColor;
     import mx.charts.LegendItem;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
      {Expense:"Taxes", April:2000, May:321, June:131, July:1100, August:200, September:1400,
October:42},
        {Expense:"Rent", April:1000, May:95, June:313, July:600, August:400, September:200,
October:52},
        {Expense:"Taxes", April:2000, May:321, June:131, July:90, August:500, September:900,
October:300},
       {Expense:"Bills", April:100, May:478, June:841, July:400, August:600, September:1100,
October:150}
     ]);

     [Bindable]
     private var rowSize:int = 5;

     [Bindable]
     private var rowSizes:ArrayCollection = new ArrayCollection(
          [ {label:"1 Per Row", data:1},
            {label:"2 Per Row", data:2},
            {label:"3 Per Row", data:3},
            {label:"4 Per Row", data:4},
            {label:"5 Per Row", data:5},
            {label:"6 Per Row", data:6},
            {label:"7 Per Row", data:7} ]);

     private function drawLegend():void {
          clearLegend();
          // Use a counter for the series.
          var z:int = 0;
          var numRows:int;
          if (myChart.series.length % rowSize == 0) {
               // The number of series is exactly divisible by the rowSize.
               numRows = Math.floor(myChart.series.length / rowSize);
          } else {
               // One extra row is needed if there is a remainder.
               numRows = Math.floor(myChart.series.length / rowSize) + 1;
          }
          for (var j:int = 0; j < numRows; j++) {
          var gr:GridRow = new GridRow();
          myGrid.addChild(gr);

          for (var k:int = 0; k < rowSize; k++) {
               // As long as the series counter is less than the number of series...
               if (z < myChart.series.length) {
                    var gi:GridItem = new GridItem();
                    gr.addChild(gi);

                    var li:LegendItem = new LegendItem();

                    // Apply the current series' displayName to the LegendItem's label.
                    li.label = myChart.series[z].displayName;

                    // Get the current series' fill.
                    var sc:SolidColor = myChart.series[z].getStyle("fill");

                    // Apply the current series' fill to the corresponding LegendItem.
                    li.setStyle("fill", sc);

                    // Apply other styles to make the LegendItems look uniform.
                    li.setStyle("textIndent", 5);
                    li.setStyle("labelPlacement", "left");
                    li.setStyle("fontSize", 9);

                    gi.setStyle("backgroundAlpha", "1");
                    gi.setStyle("backgroundColor", sc.color);
                    gi.width = 80;

                    // Add the LegendItem to the GridItem.
                    gi.addChild(li);

                    // Increment any time a LegendItem is added.
                    z++;
               }
          }
          }
     }

     private function clearLegend():void {
          myGrid.removeAllChildren();
     }

     private function changeRowSize(e:Event):void {
          rowSize = ComboBox(e.target).selectedItem.data;
          drawLegend();
     }
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Custom Legend in Grid" width="500" height="750">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:BarChart id="myChart" dataProvider="{expenses}" height="400" showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis dataProvider="{expenses}" categoryField="Expense"/>
        </mx:verticalAxis>
        <mx:series>
```

```
            <mx:BarSeries xField="April" displayName="April"/>
            <mx:BarSeries xField="May" displayName="May"/>
            <mx:BarSeries xField="June" displayName="June"/>
            <mx:BarSeries xField="July" displayName="July"/>
            <mx:BarSeries xField="August" displayName="August"/>
            <mx:BarSeries xField="September" displayName="September"/>
            <mx:BarSeries xField="October" displayName="October"/>
         </mx:series>
      </mx:BarChart>
      <mx:HRule width="100%"/>
      <mx:Grid id="myGrid"/>
      <mx:HRule width="100%"/>
      <s:HGroup>
           <s:Label text="Select the number of rows in the Legend."/>
           <s:ComboBox id="cb1"
                dataProvider="{rowSizes}"
                close="changeRowSize(event)" selectedIndex="4"/>
      </s:HGroup>
   </s:Panel>
</s:Application>
```

Note that in this example, you use the `addChild()` method to add the Legend to the Grid container. MX containers use the `addChild()` method, while Spark containers use the `addElement()` method.

Events and effects in charts

You can use Adobe® Flex® charting controls to make interesting and engaging charts. Important factors to consider are how users' interactions with your applications trigger effects and events.

Handling user interactions with charts

Chart controls accept many kinds of user interactions, from moving the mouse over a data point to clicking or double-clicking on that data point. You can create an event handler for each of these interactions and use the Event object in that handler to access the data related to that interaction. For example, if a user clicks on a column in a ColumnChart control, you can access that column's data in the click event handler of that chart.

The chart controls support the mouse events that are inherited from the UIComponent class: `mouseMove`, `mouseOver`, `mouseUp`, `mouseDown`, and `mouseOut`. These events are of type MouseEvent. In addition, the base class for all chart controls, ChartBase, adds support for the the ChartEvent and ChartItemEvent events.

The ChartItemEvent class defines events that are specific to the chart components, such as when the user clicks a data item in a chart. The ChartEvent class defines events that occur when the user clicks or double-clicks on a chart control, but not on a chart item in that chart control.

The following table describes the ChartEvent event types:

Chart event type	**Description**
`chartClick`	Broadcast when the user clicks the mouse button while the mouse pointer is over a chart but not on a chart item.
`chartDoubleClick`	Broadcast when the user double-clicks the mouse button while the mouse pointer is over a chart but not on a chart item.

The following table describes the ChartItemEvent event types:

Chart data event type	Description
`change`	Dispatched when the selected item or items changes in the chart.
`itemClick`	Broadcast when the user clicks the mouse button while over a data point.
`itemDoubleClick`	Broadcast when the user double-clicks the mouse button while over a data point.
`itemMouseDown`	Broadcast when the mouse button is down while over a data point.
`itemMouseMove`	Broadcast when the user moves the mouse pointer while over a data point.
`itemMouseUp`	Broadcast when the user releases the mouse button while over a data point.
`itemRollOut`	Broadcast when the closest data point under the mouse pointer changes.
`itemRollOver`	Broadcast when the user moves the mouse pointer over a new data point.

In addition to the ChartEvent and ChartItemEvent classes, there is also the LegendMouseEvent. This class defines events that are broadcast when the user clicks on legend items or mouses over them.

About chart events

Chart events are triggered when the user clicks or double-clicks the mouse button while the mouse pointer is over a chart control, but not over a chart item in that chart control. These events are dispatched only if the hit data set is empty.

Chart events are of type ChartEvent. Because ChartEvent events are part of the charts package, and not part of the events package, you must import the appropriate classes in the mx.charts.events package to use a ChartEvent event.

The following example logs a ChartEvent when you click or double-click the chart control, but only if you are not over a chart item or within it’s range, as determined by its `mouseSensitivity` property.

```
<?xml version="1.0"?>
<!-- charts/BasicChartEvent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600" width="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>

    <fx:Script><![CDATA[
        import mx.charts.events.ChartEvent;
        private function chartEventHandler(event:ChartEvent):void {
            /*
             * The ChartEvent will only be dispatched if the mouse is _not_ over a
             * chart item or if it is outside of the range defined by the
             * mouseSensitivity property.
            */
            ta1.text += "Event of type " + event.type + " was triggered.\n";
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";

    s|Panel {
        paddingLeft:5;
        paddingRight:5;
        paddingTop:5;
        paddingBottom:5;
    }
  </fx:Style>

  <s:Panel title="Chart click events">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <s:HGroup>
         <mx:PlotChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true"
            mouseSensitivity="10"
            doubleClickEnabled="true"
            chartClick="chartEventHandler(event)"
            chartDoubleClick="chartEventHandler(event)">
            <mx:series>
               <mx:PlotSeries
                    xField="expenses"
                    yField="profit"
                    displayName="Plot 1"/>
               <mx:PlotSeries
                    xField="amount"
                    yField="expenses"
                    displayName="Plot 2"/>
               <mx:PlotSeries
                    xField="profit"
                    yField="amount"
                    displayName="Plot 3"/>
            </mx:series>
         </mx:PlotChart>
         <s:TextArea id="ta1" width="150" height="300"/>
       </s:HGroup>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

About chart data events

Chart data events are triggered only when the user moves the mouse pointer over a data point, whereas UIComponent events are triggered by any mouse interaction with a control.

The chart data events are of type ChartItemEvent. Because ChartItemEvent events are part of the charts package, and not part of the events package, you must import the appropriate classes in the mx.charts.events package to use a ChartItemEvent event.

The following example opens an alert when the user clicks a data point (a column) in the chart:

```
<?xml version="1.0"?>
<!-- charts/DataPointAlert.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
      <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
      <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
      <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.controls.Alert;
     import mx.charts.events.ChartItemEvent;
     public function myHandler(e:ChartItemEvent):void {
        Alert.show("Chart data was clicked");
     }
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Clickable Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        itemClick="myHandler(event)"
        dataProvider="{srv.lastResult.data.result}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries yField="expenses" displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
</s:Application>
```

Using the HitData object

Flex dispatches a ChartItemEvent object for each chart data event such as when you click on an item in the series. In addition to the standard `target` and `type` properties that all Event objects have, Flex adds the `hitData` property to the ChartItemEvent object. This property is an object instance of the HitData class. The `hitData` property contains information about the data point that is closest to the mouse pointer at the time of the mouse event.

The item property of the HitData object refers to the data point. You can use that property to access its value by its name, as the previous example shows. The HitData*x* and *y* properties refer to the screen coordinates of the data point.

Only data points within the radius determined by the mouseSensitivity property can trigger an event on that data point. For more information, see “Changing mouse sensitivity” on page 1335.

The following example uses the itemDoubleClick event handler to display HitData information for data points in a column chart when the user clicks on a column. Because each column in the ColumnChart control is associated with a single data point value, clicking the mouse anywhere in a column displays the same information for the x and y coordinates.

```
<?xml version="1.0"?>
<!-- charts/HitDataOnClick.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();init()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.charts.events.ChartItemEvent;
     // Define the event handler.
     public function myListener(e:ChartItemEvent):void {
        ti1.text = e.hitData.item.expenses;
        ti2.text = String(e.hitData.x) + ", " + String(e.hitData.y);
     }
     // Define event listeners when the application initializes.
     public function init():void {
        myChart.addEventListener(ChartItemEvent.ITEM_DOUBLE_CLICK, myListener);
     }
  ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Accessing HitData Object in Event Handlers">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        doubleClickEnabled="true"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries yField="expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <s:Form width="100%">
        <!--Define a form to display the event information.-->
        <s:FormItem label="Expenses:">
           <s:TextInput id="ti1" width="100%"/>
        </s:FormItem>
        <s:FormItem label="x/y:">
           <s:TextInput id="ti2" width="100%"/>
        </s:FormItem>
     </s:Form>
  </s:Panel>
</s:Application>
```

Getting chart elements

The HitData object accesses the chart's ChartItem objects. These objects represent data points on the screen. In addition to providing access to the data of data points, ChartItem objects provide information about the size and position of graphical elements that make up the chart. For example, you can get the *x* and *y* positions of columns in a ColumnChart.

The following example uses a semitransparent Canvas container to highlight the data point that the user clicks on with the mouse. The application also accesses the ChartItem object to get the current value to display in a ToolTip on that Canvas:

```
<?xml version="1.0"?>
<!-- charts/ChartItemObjectAccess.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();init()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|ToolTip {
        fontSize:24;
     }
     mx|ColumnChart {
        gutterLeft: 54;
     }
```

```
</fx:Style>
<fx:Script><![CDATA[
   import mx.core.IFlexDisplayObject;
   import mx.charts.events.ChartItemEvent;
   import mx.charts.series.items.ColumnSeriesItem;
   import mx.charts.series.ColumnSeries;
   import mx.effects.Move;
   import mx.charts.HitData;
   public var m:mx.effects.Move;
   public var hitData:mx.charts.HitData;
   public var chartItem:ColumnSeriesItem;
   public var renderer:IFlexDisplayObject;
   public var series:ColumnSeries;
   public var chartItemPoint:Point;
   public var highlightBoxPoint:Point;
   public var leftAdjust:Number;
   private function init():void {
      m = new Move(highlightBox);
      /* This is used to adjust the x location of
         highlightBox to account for the left gutter width. */
      leftAdjust = myChart.getStyle("gutterLeft") - 14;
      trace("leftAdjust: " + leftAdjust);
   }
   private function overData(event:ChartItemEvent):void {
      hitData = event.hitData;
      chartItem = ColumnSeriesItem(hitData.chartItem);
      renderer = chartItem.itemRenderer;
      series = ColumnSeries(hitData.element);
      /* Add 10 pixels to give it horizontal overlap. */
      highlightBox.width = renderer.width * 2 + 10;
      /* Add 20 pixels to give it vertical overlap. */
      highlightBox.height = renderer.height + 20;
      highlightBoxPoint = new Point(highlightBox.x,
          highlightBox.y);
      /* Convert the ChartItem's pixel values from local
         (relative to the component) to global (relative
         to the stage). This enables you to place the Canvas
         container using the x and y values of the stage. */
      chartItemPoint = myChart.localToGlobal(new Point(chartItem.x, chartItem.y));
      /* Define the parameters of the move effect and play the effect. */
      m.xTo = chartItemPoint.x + leftAdjust;
      m.yTo = chartItemPoint.y;
      m.duration = 500;
      m.play();
      highlightBox.toolTip = "$" + chartItem.yValue.toString();
   }
]]></fx:Script>
  <s:layout>
      <s:VerticalLayout/>
  </s:layout>
<s:Panel id="p1" title="Highlighted Columns">
      <s:layout>
          <s:VerticalLayout/>
      </s:layout>
   <mx:ColumnChart id="myChart"
      dataProvider="{srv.lastResult.data.result}"
      itemClick="overData(event)"
```

```
            mouseSensitivity="0">
            <mx:horizontalAxis>
               <mx:CategoryAxis categoryField="month"/>
            </mx:horizontalAxis>
            <mx:series>
               <mx:ColumnSeries
                    displayName="Expenses"
                    yField="expenses"/>
               <mx:ColumnSeries
                    displayName="Profit"
                    yField="profit"/>
            </mx:series>
         </mx:ColumnChart>
         <mx:Legend dataProvider="{myChart}"/>
   </s:Panel>
   <!-- Define the canvas control that will be used as a highlight. -->
   <mx:Canvas id="highlightBox"
            y="0" x="0"
            backgroundColor="0xFFFF00"
            alpha=".5"/>
</s:Application>
```

For information about changing the appearance of ChartItem objects, see “Skinning ChartItem objects” on page 1232.

Getting data with coordinates

When you create charts, you can use the coordinates on the screen to get the nearest data point or data points, or conversely, to pass the data point and get the coordinates.

The `findDataPoints()` and `localToData()` methods take coordinates and return data. The `findDataPoints()` method returns a HitData object. The `localToData()` method returns an Array of the data.

You can also pass the data itself and get the coordinates with the `dataToLocal()` method.

Using the findDataPoints() method

You can use the chart control’s `findDataPoints()` method to get an Array of HitData objects by passing in x and y coordinates. If the coordinates do not correspond to the location of a data point, the `findDataPoints()` method returns `null`. Otherwise, the `findDataPoints()` method returns an Array of HitData objects.

The `findDataPoints()` method has the following signature:

```
findDataPoints(x:Number, y:Number):Array
```

The following example creates a PlotChart control and records the location of the mouse pointer as the user moves the mouse over the chart. It uses the `findDataPoints()` method to get an Array of HitData objects, and then displays some of the first object’s properties.

```
<?xml version="1.0"?>
<!-- charts/FindDataPoints.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.charts.HitData;
     public function handleMouseMove(e:MouseEvent):void {
        // Use coordinates to get HitData object of
        // current data point.
        var hda:Array =
            chart.findDataPoints(e.currentTarget.mouseX,
            e.currentTarget.mouseY);
        if (hda[0]) {
           ta.text = "Found data point " +
                hda[0].chartItem.index + " (x/y):" +
                Math.round(hda[0].x) + "," +
                Math.round(hda[0].y) + "\n";
           ta.text += "Expenses:" + hda[0].item.expenses;
        } else {
           ta.text = "No data point found  (x/y):" +
                Math.round(e.currentTarget.mouseX) +
                "/" + Math.round(e.currentTarget.mouseY);
        }
     }
  ]]></fx:Script>
    <s:layout>
```

```
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Plot Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PlotChart id="chart"
        mouseMove="handleMouseMove(event)"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        mouseSensitivity="5">
        <mx:series>
           <mx:PlotSeries
                xField="profit"
                yField="expenses"/>
        </mx:series>
     </mx:PlotChart>
  </s:Panel>
  <s:TextArea id="ta" width="300" height="50"/>
</s:Application>
```

Using the localToData() method

The `localToData()` method takes a Point object that represents the x and y coordinates you want to get the data for and returns an Array of data values, regardless of whether any data items are at or near that point. You call this method on the series.

The following example creates a Point object from the mouse pointer's location on the screen and displays the data values associated with that point:

```
<?xml version="1.0"?>
<!-- charts/LocalToData.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     public var p:Point;
     private function updateDetails(e:MouseEvent):void {
        p = new Point(myChart.mouseX, myChart.mouseY);
        mpos.text = "(" + p.x + "," + p.y + ")";
        var d:Array = series1.localToData(p);
        var da:Date = new Date(d[0]);

        // Round y coordinate value on column to two places.
        var v:Number = int((d[1])*100)/100;

        dval.text = "(" + daxis.formatForScreen(da) + ", " + v + ")";
        p = series1.dataToLocal(d[0], d[1]);
        dpos.text ="(" + Math.floor(p.x) + ", " + Math.floor(p.y) + ")";
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        dataProvider="{srv.lastResult.data.result}"
        mouseMove="updateDetails(event)"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:DateTimeAxis id="daxis" dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries id="series1"
                xField="date"
                yField="close"
                displayName="Close"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
  <s:Form width="300">
     <s:FormItem label="Mouse Position:">
        <s:Label id="mpos"/>
     </s:FormItem>
     <s:FormItem label="Data Position:">
        <s:Label id="dpos"/>
     </s:FormItem>
     <s:FormItem label="Data:">
        <s:Label id="dval"/>
     </s:FormItem>
  </s:Form>
</s:Application>
```

The chart type determines how coordinates are mapped, and how many values are returned in the Array. The values returned are typically numeric values.

In a chart that is based on the CartesianChart class (for example, a BarChart or ColumnChart control), the first item in the returned Array is the value of the *x*-coordinate along the horizontal axis, and the second item is the value of the *y*-coordinate along the vertical axis.

In a chart based on the PolarChart class (such as PieChart), the returned Array maps the coordinates to a set of polar coordinates—an angle around the center of the chart, and a distance from the center. Those values are mapped to data values that use the first (angular) and second (radial) axes of the chart.

Using the dataToLocal() method

The `dataToLocal()` method converts a set of values tox and y coordinates on the screen. The values you give the method are in the "data space" of the chart; this method converts these values to coordinates. The *data space* is the collection of all possible combinations of data values that a chart can represent.

The number and meaning of arguments passed to the `dataToLocal()` method depend on the chart type. For CartesianChart controls, such as the BarChart and ColumnChart controls, the first value is used to map along the x axis, and the second value is used to map along the y axis.

For PolarChart controls, such as the PieChart control, the first value maps to the angle around the center of the chart, and the second value maps to the distance from the center of the chart along the radius.

The coordinates returned are based on 0,0 being the upper-left corner of the chart. For a ColumnChart control, for example, the height of the column is inversely related to the x coordinate that is returned.

Changing mouse sensitivity

You use the `mouseSensitivity` property of the chart control to determine when the mouse pointer is considered to be over a data point; for example:

```
<mx:ColumnChart id="chart" dataProvider="{dataSet}" mouseSensitivity="30">
```

The current data point is the nearest data point to the mouse pointer that is less than or equal to the number of pixels that the `mouseSensitivity` property specifies.

The default value of the `mouseSensitivity` property is 3 pixels. If the mouse pointer is 4 or more pixels away from a data point, Flex does not trigger a ChartItemEvent (of types such as `itemRollOver` or `itemClick`). Flex still responds to events such as `mouseOver` and `click` by generating a ChartEvent object of type `chartClick` or `chartDoubleClick`.

The following example initially sets the `mouseSensitivity` property to 20, but lets the user change this value with the HSlider control:

```
<?xml version="1.0"?>
<!-- charts/MouseSensitivity.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Mouse Sensitivity">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:PlotChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
        mouseSensitivity="{mySlider.value}">
        <mx:series>
           <mx:PlotSeries
                xField="expenses"
                yField="profit"
                displayName="Plot 1"/>
           <mx:PlotSeries
                xField="amount"
                yField="expenses"
                displayName="Plot 2"/>
           <mx:PlotSeries
                xField="profit"
                yField="amount"
                displayName="Plot 3"/>
        </mx:series>
     </mx:PlotChart>
     <s:HGroup>
         <mx:Legend dataProvider="{myChart}"/>
         <s:VGroup>
             <s:Label text="Mouse Sensitivity:"/>
             <mx:HSlider id="mySlider"
                minimum="0"
                maximum="300"
                value="20"
                dataTipPlacement="top"
                tickColor="black"
                snapInterval="1"
                tickInterval="20"
                labels="['0','300']"
                allowTrackClick="true"
                liveDragging="true"/>
        </s:VGroup>
      </s:HGroup>
  </s:Panel>
</s:Application>
```

You can use the `mouseSensitivity` property to increase the area that triggers DataTip events or emits chart-related events. If the mouse pointer is within the range of multiple data points, Flex chooses the closest data point. For DataTip events, if you have multiple DataTip controls enabled, Flex displays all DataTip controls within range. For more information, see “Showing multiple DataTip objects” on page 1292.

Disabling interactivity

You can make the series in a chart ignore all mouse events by setting the `interactive` property to `false` for that series. The default value is `true`. This lets you disable mouse interactions for one series while allowing it for another.

The following example disables interactivity for events on the first and third PlotSeries objects:

```
<?xml version="1.0"?>
<!-- charts/DisableInteractivity.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Disable Interactivity">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:PlotChart id="myChart" 
        dataProvider="{srv.lastResult.data.result}" 
        showDataTips="true">
        <mx:series>
           <mx:PlotSeries 
                xField="expenses" 
                yField="profit" 
                displayName="Plot 1" 
                interactive="false"/>
           <mx:PlotSeries 
                xField="amount" 
                yField="expenses" 
                displayName="Plot 2"/>
           <mx:PlotSeries 
                xField="profit" 
                yField="amount" 
                displayName="Plot 3" 
                interactive="false"/>
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Disabling the series interactivity has the following results:

- The series does not show DataTip controls.
- The series does not generate a `hitData` structure on any chart data event.
- The series does not return a `hitData` structure when you call the `findDataPoints()` method on the chart.

Using effects with charts

Chart controls support the standard Flex effects such as Zoom and Fade. You can use these effects to make the entire chart control zoom in or fade out in your applications. In addition, chart data series support the following effect classes that apply to the data in the chart:

- SeriesInterpolate
- SeriesSlide
- SeriesZoom

These effects zoom or slide the chart items inside the chart control. These classes are in the mx.charts.effects.effects.* package. The base class for chart effects is SeriesEffect.

All chart controls and series support the standard Flex triggers and effects that are inherited from UIComponent. These triggers include `focusInEffect`, `focusOutEffect`, `moveEffect`, `showEffect`, and `hideEffect`. The charting controls also include the `showDataEffect` and `hideDataEffect` triggers.

For information on creating complex effects, see "Introduction to effects" on page 1784.

Using standard effect triggers

Chart controls support standard effects triggers, such as `showEffect` and `hideEffect`.

The following example defines a set of wipe effects that Flex executes when the user toggles the chart's visibility by using the Button control. Also, Flex fades in the chart when it is created.

```
<?xml version="1.0"?>
<!-- charts/StandardEffectTriggers.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
          <!-- Define the effects -->
          <mx:Parallel id="showEffects">
             <mx:WipeRight duration="2000"/>
             <mx:Fade alphaFrom="0" alphaTo="1" duration="4000"/>
          </mx:Parallel>
          <mx:Parallel id="hideEffects">
             <mx:Fade alphaFrom="1" alphaTo="0" duration="2500"/>
             <mx:WipeLeft duration="3000"/>
          </mx:Parallel>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Area Chart with Effects">
     <s:layout>
```

```
            <s:HorizontalLayout/>
        </s:layout>
        <mx:AreaChart id="myChart"
           dataProvider="{srv.lastResult.data.result}"
           creationCompleteEffect="showEffects"
           hideEffect="hideEffects"
           showEffect="showEffects">
           <mx:horizontalAxis>
              <mx:CategoryAxis categoryField="month"/>
           </mx:horizontalAxis>
           <mx:series>
              <mx:AreaSeries yField="profit" displayName="Profit"/>
              <mx:AreaSeries yField="expenses" displayName="Expenses"/>
           </mx:series>
        </mx:AreaChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
    <s:Button label="Toggle visibility"
      click="myChart.visible=!myChart.visible"/>
</s:Application>
```

A negative aspect of using standard effect triggers with charts is that the effects are applied to the entire chart control, and not just the data in the chart control. The result is that an effect such as Fade causes the chart's axes, gridlines, labels, and other chart elements, in addition to the chart's data, to fade in or out during the effect. To solve this, you use special charting effect triggers (see "Using charting effect triggers" on page 1339).

Using charting effect triggers

Charts have two unique effect triggers: `hideDataEffect` and `showDataEffect`. You set these triggers on the data series for the chart. Whenever the data for a chart changes, Flex executes these triggers in succession on the chart's data. The chart control's other elements, such as gridlines, axis lines, and labels are not affected by the effect.

The `hideDataEffect` trigger defines the effect that Flex uses as it hides the current data from view. The `showDataEffect` trigger defines the effect that Flex uses as it moves the current data into its final position on the screen.

Because Flex triggers the effects associated with `hideDataEffect` and `showDataEffect` when the data changes, there is "old" data and "new" data. The effect associated with the `hideDataEffect` trigger is the "old" data that will be replaced by the new data.

The order of events is:

1. Flex first uses the `hideDataEffect` trigger to invoke the effect set for each element of a chart that is about to change. These triggers execute at the same time.
2. Flex then updates the chart with its new data. Any elements (including the grid lines and axes) that do not have effects associated with them are updated immediately with their new values.
3. Flex then invokes the effect set with the `showDataEffect` trigger for each element associated with them. Flex animates the new data in the chart.

Charting effects with data series

Three effects classes are unique to charting: SeriesInterpolate, SeriesSlide, and SeriesZoom. You use these effects on a data series to achieve an effect when the data in that series changes. These effects can have a great deal of visual impact. Data series do not support other Flex effects.

The following example shows the SeriesSlide effect making the data slide in and out of a screen when the data changes. You can trigger changes to data in a chart series in many ways. Most commonly, you trigger an effect when the data provider changes for a chart control. In the following example, when the user clicks the button, the chart control's data provider changes, triggering the effect:

```
<?xml version="1.0"?>
<!-- charts/BasicSeriesSlideEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_fred.send();srv_strk.send()"
    height="600">
    <fx:Declarations>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/stocks.aspx -->
         <!-- View source of the following pages to see the structure of the data that Flex
uses in this example. -->
       <mx:HTTPService id="srv_fred" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=FRED"/>
       <mx:HTTPService id="srv_strk" url="http://aspexamples.adobe.com/chart_examples/stocks-
xml.aspx?tickerSymbol=STRK"/>
        <!-- Define chart effects -->
        <mx:SeriesSlide
            id="slideIn"
            duration="1000"
            direction="up"/>
        <mx:SeriesSlide
            id="slideOut"
            duration="1000"
            direction="down"/>
    </fx:Declarations>
    <fx:Script><![CDATA[
        public function changeProvider():void {
            if (series1.displayName == "STRK") {
                myChart.dataProvider = srv_fred.lastResult.data.result;
                series1.displayName = "FRED";
                lbl.text = "Ticker: " + series1.displayName;
            } else {
                myChart.dataProvider = srv_strk.lastResult.data.result;
                series1.displayName = "STRK";
                lbl.text = "Ticker: " + series1.displayName;
            }
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Column Chart with Basic Series Slide Effect"
        height="493">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Label id="lbl" text="Ticker: FRED"
            width="233.5" height="44"
            fontSize="24" color="#091D96"
            top="5" right="10"/>
        <mx:ColumnChart id="myChart"
            dataProvider="{srv_fred.lastResult.data.result}">
            <mx:horizontalAxis>
                <mx:DateTimeAxis dataUnits="days"/>
            </mx:horizontalAxis>
            <mx:verticalAxis>
                <mx:LinearAxis minimum="0" maximum="100"/>
            </mx:verticalAxis>
            <mx:series>
                <mx:ColumnSeries id="series1"
                    xField="date" yField="close"
                    displayName="FRED"
                    showDataEffect="slideIn"
                    hideDataEffect="slideOut"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}" bottom="0"/>
    </s:Panel>
    <s:Button id="b1" click="changeProvider()" label="Toggle Ticker"/>
</s:Application>
```

This example explicitly defines the minimum and maximum values of the vertical axis. If not, Flex would recalculate these values when the new data provider was applied. The result could be a change in the axis labels during the effect.

Changing a data provider first triggers the `hideDataEffect` effect on the original data provider, which causes that data provider to "slide out," and then triggers the `showDataEffect` effect on the new data provider, which causes that data provider to "slide in."

Note: *If you set the data provider on the series and not the chart control, you must change it on the series and not the chart control.*

Another trigger of data effects is when a new data point is added to a series. The following example triggers the `showDataEffect` when the user clicks the button and adds a new item to the series' data provider:

```
<?xml version="1.0"?>
<!-- charts/AddItemEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

     <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/widgets-
xml.aspx" result="resultHandler(event)"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/widgets.aspx -->
        <!-- Define chart effect -->
        <mx:SeriesSlide id="slideIn"
            duration="1000"
            direction="up"
        />
    </fx:Declarations>
    <s:layout>
```

```
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;
          [Bindable]
          public var items:ArrayCollection;

        public function addDataItem():void {
            // Add a randomly generated value to
            // the data provider.
            var n:Number = Math.random() * 3000;
            var o:Object = {item: n};
            items.addItem(o);
        }

        private function resultHandler(e:Event):void {
          items = ArrayCollection(srv.lastResult.data.result);
        }
    ]]></fx:Script>
    <s:Panel title="Column Chart with Series Effect">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart" dataProvider="{items}">
            <mx:series>
                <mx:ColumnSeries yField="item"
                    displayName="Quantity"
                    showDataEffect="slideIn"/>
            </mx:series>
        </mx:ColumnChart>
    </s:Panel>
    <s:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</s:Application>
```

This example casts the results of the HTTPService call to an ArrayCollection, so that you can add new items with the `addItem()` method. The new items are not added to the data on the server, but just in the local application's results ArrayCollection. For more information about changing charting data at run time, see "Changing chart data at run time" on page 1115.

The charting effects have several properties in common that traditional effects do not have. All of these properties are optional. The following table lists the common properties of the charting effects:

Property	**Description**
`duration`	The amount of time, in milliseconds, that Flash Player takes to complete the entire effect. This property defines the speed with which the effect executes. The `duration` property acts as a minimum duration. The effect can take longer based on the settings of other properties. The default value is 500.
`elementOffset`	The amount of time, in milliseconds, that Flash Player delays the effect on each element in the series. Set the `elementOffset` property to 0 to make all elements start at the same time and end at the same time. Set the `elementOffset` property to an integer such as 30 to stagger the effect on each element by that amount of time. For example, with a slide effect, the first element slides in immediately. The next element begins 30 milliseconds later, and so on. The amount of time for the effect to execute is the same for each element, but the overall duration of the effect is longer. Set the `elementOffset` property to a negative value to make the effect display the last element in the series first. The default value is 20.
`minimumElementDuration`	The amount of time, in milliseconds, that an individual element should take to complete the effect. Charts with a variable number of data points in the series cannot reliably create smooth effects with only the `duration` property. For example, an effect with a duration of 1000 and `elementOffset` of 100 takes 900 milliseconds per element to complete if you have two elements in the series. This is because the start of each effect is offset by 100, and each effect finishes in 1000 milliseconds. If there are four elements in the series, each element takes 700 milliseconds to complete (the last effect starts 300 milliseconds after the first and must be completed within 1000 milliseconds). With 10 elements, each element has only 100 milliseconds to complete the effect. The value of the `minimumElementDuration` property sets a minimal duration for each element. No element of the series takes less than this amount of time (in milliseconds) to execute the effect. As a result, it is possible for an effect to take longer than a specified duration if you specify at least two of the following three properties: `duration`, `offset`, and `minimumElementDuration`. The default value is 0, which means that the `duration` property is used to control the total length of time, in milliseconds, that Flex takes to complete the effect.
`offset`	The amount of time, in milliseconds, that Flash Player delays the start of the effect. Use this property to stagger effects on multiple series. The default value is 0.

Using the SeriesSlide effect

The SeriesSlide effect slides a data series into and out of the chart's boundaries. The SeriesSlide effect takes a `direction` property that defines the location from which the series slides. Valid values of `direction` are `left`, `right`, `up`, or `down`.

If you use the SeriesSlide effect with a `hideDataEffect` trigger, the series slides from the current position on the screen to a position off the screen, in the direction indicated by the `direction` property. If you use SeriesSlide with a `showDataEffect` trigger, the series slides from off the screen to a position on the screen, in the indicated direction.

When you use the SeriesSlide effect, the entire data series disappears from the chart when the effect begins. The data then reappears based on the nature of the effect. To keep the data on the screen at all times during the effect, you can use the SeriesInterpolate effect. For more information, see "Using the SeriesInterpolate effect" on page 1347.

The following example creates an effect called `slideDown`. Each element starts its slide 30 milliseconds after the element before it, and takes at least 20 milliseconds to complete its slide. The entire effect takes at least 1 second (1000 milliseconds) to slide the data series down. Flex invokes the effect when it clears old data from the chart and when new data appears.

```
<?xml version="1.0"?>
<!-- charts/CustomSeriesSlideEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/widgets-
xml.aspx" result="resultHandler(event)"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/widgets.aspx -->
          <!-- Define chart effect -->
          <mx:SeriesSlide duration="1000"
            direction="down"
            minimumElementDuration="20"
            elementOffset="30"
            id="slideDown"
          />
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var items:ArrayCollection;
     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item:n};
        items.addItem(o);
     }
     private function resultHandler(e:Event):void {
          items = ArrayCollection(srv.lastResult.data.result);
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart with Custom Series Slide Effect">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="slideDown"
            hideDataEffect="slideDown"/>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
  <s:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</s:Application>
```

Using the SeriesZoom effect

The SeriesZoom effect implodes and explodes chart data into and out of the focal point that you specify. As with the SeriesSlide effect, whether the effect is zooming to or from this point depends on whether it's used with a `showDataEffect` or `hideDataEffect` trigger.

The SeriesZoom effect can take several properties that define how the effect acts. The following table describes these properties:

Property	Description
`horizontalFocus` `verticalFocus`	Defines the location of the focal point of the zoom. You combine the `horizontalFocus` and `verticalFocus` properties to define the point from which the data series zooms in and out. For example, set the `horizontalFocus` property to `left` and the `verticalFocus` property to `top` to have the series data zoom to and from the upper-left corner of either element or the chart (depending on the setting of the `relativeTo` property). Valid values of the `horizontalFocus` property are `left`, `center`, `right`, and `undefined`. Valid values of the `verticalFocus` property are `top`, `center`, `bottom`, and `undefined`. If you specify only one of these two properties, the focus is a horizontal or vertical line rather than a point. For example, when you set the `horizontalFocus` property to `left`, the element zooms to and from a vertical line along the left edge of its bounding box. Setting the `verticalFocus` property to `center` causes the element to zoom to and from a horizontal line along the middle of its bounding box. The default value for both properties is `center`.
`relativeTo`	Controls the bounding box used to calculate the focal point of the zooms. Valid values for `relativeTo` are `series` and `chart`. Set the `relativeTo` property to `series` to zoom each element relative to itself. For example, each column of a ColumnChart zooms from the upper-left of the column. Set the `relativeTo` property to `chart` to zoom each element relative to the chart area. For example, each column zooms from the upper-left of the axes, the center of the axes, and so on.

The following example zooms in the data series from the upper-right corner of the chart. While zooming in, Flex displays the last element in the series first because the `elementOffset` value is negative.

```
<?xml version="1.0"?>
<!-- charts/SeriesZoomEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/widgets-
xml.aspx" result="resultHandler(event)"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/widgets.aspx -->
      <!-- Define chart effects -->
      <mx:SeriesZoom id="zoomOut"
         duration="2000"
         minimumElementDuration="50"
         elementOffset="50"
         verticalFocus="top"
         horizontalFocus="left"
         relativeTo="chart"/>
      <mx:SeriesZoom id="zoomIn"
         duration="2000"
         minimumElementDuration="50"
         elementOffset="-50"
         verticalFocus="top"
         horizontalFocus="right"
         relativeTo="chart"/>
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var items:ArrayCollection;
     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
     private function resultHandler(e:Event):void {
          items = ArrayCollection(srv.lastResult.data.result);
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart with Series Zoom Effect">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="zoomIn"
            hideDataEffect="zoomOut"/>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
  <s:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</s:Application>
```

When you use the SeriesZoom effect, the entire data series disappears from the chart when the effect begins. The data then reappears based on the nature of the effect. To have the data stay on the screen at all times during the effect, you can use the SeriesInterpolate effect. For more information, see “Using the SeriesInterpolate effect” on page 1347.

Using the SeriesInterpolate effect

The SeriesInterpolate effect moves the graphics that represent the existing data in the series to the new points. Instead of clearing the chart and then repopulating it as with SeriesZoom and SeriesSlide, this effect keeps the data on the screen at all times.

You use the SeriesInterpolate effect only with the `showDataEffect` effect trigger. It has no effect if set with a `hideDataEffect` trigger.

The following example sets the `elementOffset` property of the SeriesInterpolate effect to 0. As a result, all elements move to their new locations without disappearing from the screen.

```
<?xml version="1.0"?>
<!-- charts/SeriesInterpolateEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/widgets-
xml.aspx" result="resultHandler(event)"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/widgets.aspx -->
      <!-- Define chart effect. -->
      <mx:SeriesInterpolate id="rearrangeData"
         duration="1000"
         minimumElementDuration="200"
         elementOffset="0"
      />
    </fx:Declarations>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var items:ArrayCollection;
     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
     private function resultHandler(e:Event):void {
          items = ArrayCollection(srv.lastResult.data.result);
     }
  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Column Chart with Series Interpolate Effect">
     <s:layout>
         <s:VerticalLayout/>
     </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="rearrangeData"
           />
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
  <s:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</s:Application>
```

Applying chart series effects with ActionScript

You can define effects and apply them to chart series by using ActionScript. One way to apply an effect is the same as the way you apply style properties. You use the `setStyle()` method and Cascading Style Sheets (CSS) to apply the effect.

For more information on using styles, see “Styles and themes” on page 1492.

The following example defines the `slideIn` effect that plays every time the user adds a data item to the chart control’s ColumnSeries. The effect is applied to the series by using the `setStyle()` method when the application first loads.

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsAsStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();srv.send();"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/widgets-
xml.aspx" result="resultHandler(event)"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/widgets.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;
     [Bindable]
     public var items:ArrayCollection;
     public var rearrangeData:SeriesInterpolate;
     public function initApp():void {
        rearrangeData = new SeriesInterpolate();
        rearrangeData.duration = 1000;
        rearrangeData.minimumElementDuration = 200;
        rearrangeData.elementOffset = 0;
        // Apply the effect as a style.
        mySeries.setStyle("showDataEffect", rearrangeData);
     }
     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
     private function resultHandler(e:Event):void {
          items = ArrayCollection(srv.lastResult.data.result);
     }
  ]]></fx:Script>
  <s:Panel title="Column Chart with Series Interpolate Effect">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries id="mySeries"
            yField="item" displayName="Quantity"/>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
  <s:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</s:Application>
```

When you define an effect in ActionScript, you must import the appropriate classes. If you define an effect by using MXML, the compiler imports the class for you.

Instead of applying effects with the `setStyle()` method, you can apply them with CSS if you predefine them in your MXML application; for example:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithCSS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/widgets-
xml.aspx" result="resultHandler(event)"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/widgets.aspx -->
        <!-- Define chart effect. -->
        <mx:SeriesSlide
            duration="1000"
            direction="down"
            minimumElementDuration="20"
            elementOffset="30"
            id="slideDown"/>
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";
     mx|ColumnSeries {
        showDataEffect:slideDown;
        hideDataEffect:slideDown;
     }
  </fx:Style>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var items:ArrayCollection;
     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item:n};
        items.addItem(o);
     }
     private function resultHandler(e:Event):void {
          items = ArrayCollection(srv.lastResult.data.result);
     }
  ]]></fx:Script>
  <s:Panel title="Column Chart with Custom Series Slide Effect">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries yField="item" displayName="Quantity"/>
        </mx:series>
     </mx:ColumnChart>
  </s:Panel>
  <s:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</s:Application>
```

You can use controls such as a slider to let the user adjust the effect's properties. To bind the properties of an ActionScript object, such as an effect, to a control, you use methods of the BindingUtils class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithActionScriptSlider.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();srv.send();"
    height="750">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/widgets-
xml.aspx" result="resultHandler(event)"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/widgets.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;
     import mx.binding.utils.BindingUtils;
     public var rearrangeData:SeriesInterpolate;

     [Bindable]
     public var items:ArrayCollection;
     public function initApp():void {
        rearrangeData = new SeriesInterpolate();
        // Bind effect properties to slider controls
        BindingUtils.bindProperty(rearrangeData, "duration", durationSlider, "value");
```

```
            BindingUtils.bindProperty(rearrangeData, "minimumElementDuration",
                minimumElementDurationSlider, "value");
            BindingUtils.bindProperty(rearrangeData, "elementOffset",
                elementOffsetSlider, "value");
        }
        public function addDataItem():void {
            // Add a randomly generated value to the data provider.
            var n:Number = Math.random() * 3000;
            var o:Object = {item: n};
            items.addItem(o);
        }
        public function resetSliders():void {
            durationSlider.value = 1000;
            minimumElementDurationSlider.value = 200;
            elementOffsetSlider.value = 0;
        }
        private function resultHandler(e:Event):void {
              items = ArrayCollection(srv.lastResult.data.result);
        }
    ]]></fx:Script>
    <s:Panel title="Column Chart with Series Interpolate Effect">
            <s:layout>
                <s:VerticalLayout/>
            </s:layout>
        <mx:ColumnChart id="myChart" dataProvider="{items}">
            <mx:series>
                <mx:ColumnSeries id="mySeries"
                 yField="item" displayName="Quantity"
                 showDataEffect="rearrangeData"/>
            </mx:series>
        </mx:ColumnChart>
    </s:Panel>
    <s:Button id="b1" click="addDataItem()" label="Add Data Item"/>
    <s:Panel title="Effects">
        <s:Form>
            <s:FormItem label="Duration">
                <mx:HSlider id="durationSlider"
                 minimum="1"
                 maximum="10000"
                 value="1000"
                 dataTipPlacement="top"
                 tickColor="black"
                 snapInterval="500"
                 tickInterval="500"
                 labels="['0','10000']"
                 allowTrackClick="true"
                 liveDragging="true"/>
            </s:FormItem>
            <s:FormItem label="Minimum Element Duration">
                <mx:HSlider id="minimumElementDurationSlider"
                 minimum="0"
                 maximum="1000"
                 value="200"
                 dataTipPlacement="top"
                 tickColor="black"
                 snapInterval="50"
                 tickInterval="50"
                 labels="['0','1000']"
                 allowTrackClick="true"
                 liveDragging="true"/>
             </s:FormItem>
             <s:FormItem label="Element Offset">
                <mx:HSlider id="elementOffsetSlider"
                 minimum="0"
                 maximum="1000"
                 value="0"
                 dataTipPlacement="top"
                 tickColor="black"
                 snapInterval="50"
                 tickInterval="50"
                 labels="['0','1000']"
                 allowTrackClick="true"
                 liveDragging="true"/>
             </s:FormItem>
          </s:Form>
       </s:Panel>
       <s:Button id="b2" label="Reset Sliders" click="resetSliders()"/>
     </s:Application>
```

For more information about databinding in ActionScript, see “Defining data bindings in ActionScript” on page 318.

Drilling down into data

One common use of charts is to allow the user to drill down into the data. This usually occurs when the user performs some sort of event on the chart such as clicking on a wedge in a PieChart control or clicking on a column in a ColumnChart control. Clicking on a data item reveals a new chart that describes the make-up of that data item.

For example, you might have a ColumnChart control that shows the month-by-month production of widgets. To initially populate this chart, you might make a database call (through a service or some other adapter). If the user then clicks on the January column, the application could display the number of widgets of each color that were produced that month. To get the individual month’s widget data, you typically make another database call and pass a parameter to the listening service that describes the specific data you want. You can then use the resulting data provider to render the new view.

Typically, when you drill down into chart data, you create new charts in your application with ActionScript. When creating new charts in ActionScript, you must be sure to create a series, add it to the new chart’s series Array, and then call the `addElement()` method (on Spark containers) or `addChild()` method (on MX containers) to add the new chart to the display list. For more information, see “Creating charts in ActionScript” on page 1084.

One way to provide drill-down functionality is to make calls that are external to the application to get the drill-down data. You typically do this by using the chart’s `itemClick` event listener, which gives you access to the HitData object. The HitData object lets you examine what data was underneath the mouse when the event was triggered. This capability lets you perform actions on specific chart data. For more information, see “Using the HitData object” on page 1327.

You can also use a simple Event object to get a reference to the series that was clicked. The following example shows the net worth of a fictional person. When you click on a column in the initial view, the example drills down into a second view that shows the change in value of a particular asset class over time.

The following example uses the Event object to get a reference to the clicked ColumnSeries. It then drills down into the single ColumnSeries by replacing the chart’s series Array with the single ColumnSeries in the chart. When you click a column again, the chart returns to its original configuration with all ColumnSeries.

```
<?xml version="1.0"?>
<!-- charts/SimpleDrillDown.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();srv.send();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/networth-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/networth.aspx -->
    </fx:Declarations>
    <fx:Script><![CDATA[
        public var initSeriesArray:Array = new Array();
        public var level:Number = 1;
        public var newSeries:Array;

        private function initApp():void {
            // Get initial series Array -- to be reloaded when it returns
            // from a drill down.
            initSeriesArray = chart.series;
        }

        private function zoomIntoSeries(e:Event):void {
            newSeries = new Array();
            if (level == 1) {
                newSeries.push(e.currentTarget);
                level = 2;
            } else {
                newSeries = initSeriesArray;
                p1.title = "Net Worth";
                level = 1;
            }
            chart.series = newSeries;
        }

    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="p1" title="Net Worth">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
        <mx:ColumnChart id="chart"
            dataProvider="{srv.lastResult.data.result}"
            type="stacked"
            showDataTips="true">
            <mx:series>
                <mx:ColumnSeries id="s1"
                    displayName="Cash"
                    yField="cash"
                    xField="date"
                    click="zoomIntoSeries(event)"/>
                <mx:ColumnSeries id="s2"
                    displayName="Stocks"
                    yField="stocks"
                    xField="date"
                    click="zoomIntoSeries(event)"/>
                <mx:ColumnSeries id="s3"
                    displayName="Retirement"
                    yField="retirement"
                    xField="date"
                    click="zoomIntoSeries(event)"/>
                <mx:ColumnSeries id="s4"
                    displayName="Home"
                    yField="home"
                    xField="date"
                    click="zoomIntoSeries(event)"/>
                <mx:ColumnSeries id="s5"
                    displayName="Other"
                    yField="other"
                    xField="date"
                    click="zoomIntoSeries(event)"/>
            </mx:series>
            <mx:horizontalAxis >
                <mx:DateTimeAxis title="Date" dataUnits="months"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{chart}"/>
    </s:Panel>
</s:Application>
```

Another approach to drilling down into chart data is to use unused data within the existing data provider. You can do this by changing the properties of the series and axes when the chart is clicked.

The following example is similar to the previous example in that it drills down into the assets of a fictional person’s net worth. In this case, though, it shows the value of the asset classes for the clicked-on month in the drill-down view rather than the change over time of a particular asset class.

This example uses the HitData object’s `item` property to access the values of the current data provider. By building an Array of objects with the newly-discovered data, the chart is able to drill down into the series without making calls to any external services.

Drilling down into data is an ideal time to use effects such as SeriesSlide. This example also defines `seriesIn` and `seriesOut` effects to slide the columns in and out when the drilling down (and the return from drilling down) occurs.

```
<?xml version="1.0"?>
<!-- charts/DrillDownWithEffects.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();"
    height="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/networth-
xml.aspx" result="resultHandler()"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/networth.aspx -->
        <mx:SeriesSlide id="slideIn" duration="1000" direction="down"/>
        <mx:SeriesSlide id="slideOut" duration="1000" direction="up"/>
    </fx:Declarations>
    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.HitData;
        import mx.charts.events.ChartItemEvent;

        [Bindable]
        public var drillDownDataSet:ArrayCollection;
        [Bindable]
        public var dp:ArrayCollection;
          // level is a temporary variable used to determine when to drill down.
          private var level:int =  1;
        private function resultHandler():void {
          dp = ArrayCollection(srv.lastResult.data.result);
        }

        private function zoomIntoSeries(e:ChartItemEvent):void {
            if (level == 1) {
                drillDownDataSet = new ArrayCollection(genData(e));
                cs1.displayName = "Assets";
                cs1.yField = "amount";
                cs1.xField = "type";
                ca1.categoryField = "type";

                p1.title = "Asset breakdown for " + e.hitData.item.date;
                dp = drillDownDataSet;

                // Remove the Legend. It is not needed in this view because
                // each asset class is its own column in the drill down.
                p1.removeElement(myLegend);

                level = 2;
            } else {
                cs1.displayName = "All Assets";
                cs1.yField = "assets";
                cs1.xField = "date";

                ca1.categoryField = "date";
                p1.title = "Net Worth";

                // Add the Legend back to the Panel.
                p1.addElement(myLegend);

                // Reset chart to original data provider.
                resultHandler();

                level = 1;
            }
        }
          // Build an ArrayCollection of objects to make up the new data set.
        private function genData(e:ChartItemEvent):Array {
                var result:Array = [];
                trace("Cash: " + e.hitData.item.cash);
                trace("Stocks: " + e.hitData.item.stocks);
                trace("Retirement: " + e.hitData.item.retirement);
                trace("Home: " + e.hitData.item.home);
                trace("Other: " + e.hitData.item.other);

                var o1:Object = {
                    type:"cash",
                    amount:e.hitData.item.cash
                };
                var o2:Object = {
                    type:"stocks",
                    amount:e.hitData.item.stocks
                };
                var o3:Object = {
                    type:"retirement",
                    amount:e.hitData.item.retirement
                };
                var o4:Object = {
                    type:"home",
                    amount:e.hitData.item.home
                };
                var o5:Object = {
                    type:"other",
                    amount:e.hitData.item.other
                };
                trace(o1.type + ":" + o1.amount);

                result.push(o1);
                result.push(o2);
                result.push(o3);
                result.push(o4);
                result.push(o5);
                return result;
        }

    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="p1" title="Net Worth">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="chart"
            showDataTips="true"
            itemClick="zoomIntoSeries(event)"
            dataProvider="{dp}">
            <mx:series>
                <mx:ColumnSeries id="cs1"
                    displayName="All Assets"
                    yField="assets"
                    xField="date"
                    hideDataEffect="slideOut"
                    showDataEffect="slideIn"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis id="ca1" categoryField="date"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>

        <mx:Legend id="myLegend" dataProvider="{chart}"/>
    </s:Panel>
</s:Application>
```

Another way to drill down into chart data is to use the selection API. You can select a subset of data points on a chart to create a new data provider. For more information, see “Selecting chart items” on page 1358.

Selecting chart items

As part of interacting with charts, you can select data points (ChartItem objects), examine the underlying data, and then perform actions on those objects. A ChartItem class represents a single entry in the chart series’ data provider. A series is made up of an Array of ChartItem objects.

To enable data point selection, you set the value of the `selectionMode` property on the chart. Possible values of this property are `none`, `single`, and `multiple`. Setting the `selectionMode` property to `none` prevents any data points in the chart from being selected. Setting `selectionMode` to `single` lets you select one data point at a time. Setting `selectionMode` to `multiple` lets you select one or more data points at a time. The default value is `none`.

You also toggle the series’ selectability by setting the value of the `selectable` property. As a result, while you might set the `selectionMode` on the chart to `multiple`, you can make the data points in some series selectable and others not selectable within the same chart by using the series’ `selectable` property.

Methods of chart item selection

You can programmatically select data points or the user can interactively select data points in the following ways:

- Mouse selection—Move the mouse pointer over a data point and click the left mouse button to select data points. For more information, see “Keyboard and mouse selection” on page 1362.
- Keyboard selection—Use the keyboard to select one or more data points, as described in “Keyboard and mouse selection” on page 1362.
- Region selection—Draw a rectangle on the chart. This rectangle defines the range, and selects all data points within that range. For more information, see “Region selection” on page 1362.
- Programmatic selection—Programmatically select one or more data points using the chart selection API. For more information, see “Programmatic selection” on page 1363.

When selecting chart items, you should also understand the following terms:

- *caret*—The current chart item that could be selected or deselected if you press the space bar. If you select multiple items, one at a time, the caret is the last item selected. If you selected multiple items by using a range, the caret is the last item in the last series in the selection. This is not always the item that was selected last.
- *anchor*—The starting chart item for a multiple selection operation.

The following example creates three different types of charts. You can select one or more data points in each chart using the mouse, keyboard, and region selection techniques. The example displays the data points in each series that are currently selected.

```
<?xml version="1.0" ?>
<!--  charts/SimpleSelection.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv_budget.send();srv_medals.send();srv_expenses.send()"
    height="950" width="750">
    <fx:Declarations>
       <!-- View source of the following pages to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv_medals" 
url="http://aspexamples.adobe.com/chart_examples/medals-xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/medals.aspx -->
        <mx:HTTPService id="srv_budget" 
url="http://aspexamples.adobe.com/chart_examples/budget-xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/budget.aspx -->
        <mx:HTTPService id="srv_expenses" 
url="http://aspexamples.adobe.com/chart_examples/expenses-xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        private function handleChange(event:Event, ta:spark.components.TextArea):void {
            var allSeries:Array = event.currentTarget.series;
            ta.text = "";
            for (var i:int=0; i<allSeries.length; i++) {
                ta.text += "\n" + allSeries[i].id + 
                    " Selected Items: " + allSeries[i].selectedIndices;
            }
        }
    ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Bar Chart" width="100%">
         <s:layout>
             <s:HorizontalLayout/>
         </s:layout>
         <s:VGroup>
```

```
            <mx:BarChart id="myBarChart"
                height="225"
                showDataTips="true"
                dataProvider="{srv_medals.lastResult.data.result}"
                selectionMode="multiple"
                change="handleChange(event, textArea1)">
                <mx:verticalAxis>
                    <mx:CategoryAxis categoryField="country"/>
                </mx:verticalAxis>
                <mx:series>
                    <mx:BarSeries id="barSeries1"
                        yField="country"
                        xField="gold"
                        displayName="Gold"
                        selectable="true"/>
                    <mx:BarSeries id="barSeries2"
                        yField="country"
                        xField="silver"
                        displayName="Silver"
                        selectable="true"/>
                    <mx:BarSeries id="barSeries3"
                        yField="country"
                        xField="bronze"
                        displayName="Bronze"
                        selectable="true"/>
                </mx:series>
            </mx:BarChart>
            <mx:Legend dataProvider="{myBarChart}" direction="horizontal"/>
        </s:VGroup>
        <s:VGroup width="250" height="200">
              <s:Label text="Click a chart's item to select it."/>
              <s:Label text="Click and drag to select multiple items."/>
              <s:Label text="Selection Changed Event:"/>
              <s:TextArea id="textArea1" height="100" width="213"/>
          </s:VGroup>
    </s:Panel>
    <s:Panel title="Pie Chart" width="100%">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:VGroup>
            <mx:PieChart id="myPieChart"
                height="225"
                dataProvider="{srv_budget.lastResult.data.result}"
                showDataTips="true"
                selectionMode="multiple"
                change="handleChange(event, textArea2)">
                <mx:series>
                    <mx:PieSeries id="pieSeries1"
                        field="amount"
                        nameField="item"
                        labelPosition="callout"/>
                </mx:series>
            </mx:PieChart>
            <mx:Legend dataProvider="{myPieChart}" direction="horizontal"/>
        </s:VGroup>
        <s:VGroup width="250" height="200">
```

```
            <s:Label text="Click a chart's item to select it."/>
            <s:Label text="Click and drag to select multiple items."/>
            <s:Label text="Selection Changed Event:"/>
            <s:TextArea id="textArea2" height="100" width="213"/>
        </s:VGroup>
    </s:Panel>
    <s:Panel title="Plot Chart" width="100%">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:VGroup>
            <mx:PlotChart id="myPlotChart"
                height="225"
                showDataTips="true"
                dataProvider="{srv_expenses.lastResult.data.result}"
                selectionMode="multiple"
                change="handleChange(event, textArea3)">
                <mx:series>
                    <mx:PlotSeries id="plotSeries1"
                        xField="expenses"
                        yField="profit"
                        displayName="Expenses/Profit"
                        selectable="true"/>
                    <mx:PlotSeries id="plotSeries2"
                        xField="amount"
                        yField="expenses"
                        displayName="Amount/Expenses"
                        selectable="true"/>
                    <mx:PlotSeries id="plotSeries3"
                        xField="profit"
                        yField="amount"
                        displayName="Profit/Amount"
                        selectable="true"/>
                </mx:series>
            </mx:PlotChart>
            <mx:Legend dataProvider="{myPlotChart}" direction="horizontal"/>
         </s:VGroup>
         <s:VGroup width="250" height="200">
             <s:Label text="Click a chart's item to select it."/>
             <s:Label text="Click and drag to select multiple items."/>
             <s:Label text="Selection Changed Event:" />
             <s:TextArea id="textArea3" height="100" width="213"/>
         </s:VGroup>
    </s:Panel>
</s:Application>
```

You can specify the colors that are used to indicate if an item is selected, disabled, or being rolled over. You do this by using the `itemSelectionColor`, `itemDisabledColor`, and `itemRollOverColor` style properties of the chart controls. The following example specifies different colors for these states for several chart types:

```
BarChart, PieChart, PlotChart {
    itemSelectionColor: red;
    itemDisabledColor: green;
    itemRollOverColor: blue;
}
```

Region selection

Users can select all data points in an area on a chart by drawing a rectangle (the *region*) on the chart. Users define a rectangular region by clicking and holding the mouse button while they move the mouse, drawing a rectangle on the chart. On the `MOUSE_UP` event, the items inside the rectangular region are selected. The starting point for a rectangular range must be over the chart. You cannot start the rectangle outside of the chart control's bounds.

To select data points with region selection, the value of the chart's `selectionMode` property must be `multiple`. If it is `single` or `none`, then you cannot draw the selection rectangle on the chart.

For most charts, as long as any point inside the rectangle touches a data point (for example, a column in a ColumnChart control), you select that data point. For BubbleChart and PieChart controls, an item is selected only if its center is inside the selection rectangle.

Keyboard and mouse selection

To select chart items, you can use the arrow keys, space bar, and enter key on your keyboard or the mouse pointer. If you want to select more than data point, the value of the chart's `selectionMode` property must be `multiple`.

Left and right arrow keys

You use the left and right arrows to move up and down the data in the series.

Click the left arrow to select the previous item and the right arrow to select the next item in the series. When you reach the last item in a series, you move to the first item in the next series.

Up and down arrow keys

Click the up and down arrow keys to move to the next or previous series on the chart.

Click the up arrow key to select the next series in the chart's series array. Click the down arrow key to select the previous series.

The index of the item in each series remains the same. If there is only one series in the chart, then click the up arrow key to select the first data point in the series; click the down arrow key to select the last data point in the series.

Shift and control keys

You can use the shift and control keys in conjunction with the other keys to select or deselect data points and change the caret in the selection.

Hold the shift key down while clicking the right arrow key to add the next data point to the selection until you reach the end of the series. Click the left arrow key to remove the last data point from the selection.

Hold the shift key down while clicking the up arrow key to select the first item in the next series. Click the down arrow key to select the last item in the previous series.

Hold the control key down while using the arrow keys to move the caret while not deselecting the anchor item. You can then select the new caret by pressing the space bar.

Space bar

Click the space bar to toggle the selection of the caret item, if the control key is depressed. A deselected caret item appears highlighted, but the highlight color is not the same as the selected item color.

Page Up/Home and Page Down/End keys

Click the Page Up/Home and Page Down/End keys to move to the first and last items in the current series, respectively. Moving to an item makes that item the caret item, but does not select it. You can press the space bar to select it.

Mouse pointer

The default behavior of the mouse pointer is to select the data point under the mouse pointer when you click the mouse button. This also de-selects all other data points.

If you click the mouse button and drag it over a chart, you create a rectangular region that defines a selection range. All data points inside that range are selected if the `selectionMode` property is set to `multiple`.

If you select a data point, then hold the shift key down and click on another data point, you select the first point, the target point, and all points that appear inside the rectangular range that you just created. If you select a data point, then hold the control key down and click another data point, you select both data points, but not the data points in between. You can add more individual data points by continuing to hold the control key down while you select another data point.

If you click anywhere on the chart that does not have a data point without any keys held down, then you clear the selection. Clicking outside of the chart control's boundary does not clear the selection.

Programmatic selection

You can use the chart selection APIs to programmatically select one or more data points in a chart control. These APIs consist of methods and properties of the ChartBase, ChartItem, and chart series objects.

Selections with multiple items include a caret and an anchor. You can access these items by using the `caretItem` and `anchorItem` properties of the chart control.

Properties of the series

The series defines which ChartItem objects are selected. You can programmatically select items by setting the values of the following properties of the series:

- `selectedItem`
- `selectedItems`
- `selectedIndex`
- `selectedIndices`

The index properties refer to the index of the chart item in the series. This index is the same as the index in the data provider, assuming you did not sort or limit the series items.

Programmatically setting the values of these properties does not trigger a `change` event.

The following example increments and decrements the series' `selectedIndex` property to select each item, one after the other, in the series:

```
<?xml version="1.0" ?>
<!--  charts/SimplerCycle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();initApp();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;

        private function initApp():void {
            // Select the first item on start up.
            series1.selectedIndex = 0;
        }

        private function getNext(e:Event):void {
            series1.selectedIndex += 1;
        }
        private function getPrev(e:Event):void {
            series1.selectedIndex -= 1;
        }
        private function getFirst(e:Event):void {
            series1.selectedIndex = 0;
        }
        private function getLast(e:Event):void {
            series1.selectedIndex = series1.items.length - 1;
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel height="100%" width="100%">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
        <mx:ColumnChart id="myChart" 
            height="207" width="350" 
            showDataTips="true" 
            dataProvider="{srv.lastResult.data.result}" 
            selectionMode="single">
            <mx:series>
                <mx:ColumnSeries id="series1" 
                    yField="expenses" 
                    displayName="Expenses"
                    selectable="true"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
        <s:HGroup>
            <s:Label id="label0" text="Value: "/>
            <s:Label id="label1"/>
        </s:HGroup>

        <mx:Legend dataProvider="{myChart}" width="200"/>
        <s:HGroup>
            <s:Button label="|&lt;" click="getFirst(event);" />
            <s:Button label="&lt;" click="getPrev(event);" />
            <s:Button label="&gt;" click="getNext(event);" />
            <s:Button label="&gt;|" click="getLast(event);" />
        </s:HGroup>
    </s:Panel>
</s:Application>
```

To cycle through ChartItem objects in a series, you can use methods such as `getNextItem()` and `getPreviousItem()`. For more information, see “Methods and properties of the ChartBase class” on page 1369.

The `selectedIndices` property lets you select any number of ChartItems in a chart control. The following example uses the `selectedIndices` property to select all items in all series when the user presses the Ctrl+a keys on the keyboard:

```
<?xml version="1.0" ?>
<!--  charts/SelectAllItems.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();initApp();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import flash.events.KeyboardEvent;
        import mx.charts.events.ChartItemEvent;
        import mx.core.FlexGlobals;

        private function initApp():void {
          FlexGlobals.topLevelApplication.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
        }

        private function keyHandler(event:KeyboardEvent):void {
            var ctrlPressed:Boolean = event.ctrlKey;
            // If the user presses Ctrl + A, select all chart items.
            if (ctrlPressed) {
                var curKeyCode:int = event.keyCode;
                if (curKeyCode == 65) { // 65 is the keycode value for 'a'
                    selectItems();
                }
            }
        }

        private function selectItems():void {
            // Create an array of all the chart's series.
            var allSeries:Array = myChart.series;

            // Iterate over each series.
            for (var i:int=0; i<allSeries.length; i++) {
                var selectedData:Array = [];

                // Iterate over the number of items in the series.
                for (var j:int=0; j<srv.lastResult.data.result.length; j++) {
                        selectedData.push(j);
                }

                // Use the series' selectedIndices property to select all the
                // chart items.
                allSeries[i].selectedIndices = selectedData;
            }
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel height="100%" width="100%">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
        <mx:PlotChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}"
            selectionMode="multiple">
            <mx:series>
                <mx:PlotSeries id="series1"
                    xField="expenses"
                    yField="profit"
                    displayName="Expenses/Profit"
                    selectable="true"/>
                <mx:PlotSeries id="series2"
                    xField="amount"
                    yField="expenses"
                    displayName="Amount/Expenses"
                    selectable="true"/>
                <mx:PlotSeries id="series3"
                    xField="profit"
                    yField="amount"
                    displayName="Profit/Amount"
                    selectable="true"/>
            </mx:series>
        </mx:PlotChart>

        <mx:Legend dataProvider="{myChart}" width="200"/>
        <s:Label text="Click Ctrl + A to select all items."/>
    </s:Panel>
</s:Application>
```

The following example is similar to the previous example, except that it lets you set a threshold value in the TextInput control. The chart only selects chart items whose values are greater than that threshold.

```
<?xml version="1.0" ?>
<!--  charts/ConditionallySelectChartItems.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx" result="resultHandler()"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            private var expensesAC:ArrayCollection;
            private function resultHandler():void {
              expensesAC = ArrayCollection(srv.lastResult.data.result);
            }
            private function selectItems(event:Event):void {
                // Create an array of all the chart's series.
                var allSeries:Array = myChart.series;
                // Iterate over each series.
                for (var i:int=0; i<allSeries.length; i++) {
                    var selectedData:Array = [];
                    // Iterate over each item in the series.
                    for (var j:int=0; j<expensesAC.length; j++) {
                        if (expensesAC.getItemAt(j).expenses >=
                            Number(threshold.text)) {
                            selectedData.push(j);
                        }
                    }
                    // Use the series' selectedIndices property to select all the
                    // chart items that met the criteria.
                    allSeries[i].selectedIndices = selectedData;
                }
            }
        ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel height="100%" width="100%">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:PlotChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple">
            <mx:series>
                <mx:PlotSeries id="series1"
                    xField="expenses"
                    yField="profit"
                    displayName="Expenses/Profit"
                    selectable="true"/>
                <mx:PlotSeries id="series2"
                    xField="amount"
                    yField="expenses"
                    displayName="Amount/Expenses"
                    selectable="true"/>
                <mx:PlotSeries id="series3"
                    xField="profit"
                    yField="amount"
                    displayName="Profit/Amount"
                    selectable="true"/>
            </mx:series>
        </mx:PlotChart>
        <mx:Legend dataProvider="{myChart}" width="200"/>

        <s:HGroup>
            <s:Label text="'Expenses' Threshold:"/>
            <s:TextInput id="threshold" text="1500"/>
        </s:HGroup>
        <s:Button label="Select Items" click="selectItems(event)"/>

    </s:Panel>
</s:Application>
```

Methods and properties of the ChartBase class

The ChartBase class is the parent class of all chart controls. You can programmatically access ChartItem objects by using the following methods of this class:

- `getNextItem()`
- `getPreviousItem()`
- `getFirstItem()`
- `getLastItem()`

These methods return a ChartItem object, whether it is selected or not. Which object is returned depends on which one is currently selected, and which direction constant (`ChartBase.HORIZONTAL` or `ChartBase.VERTICAL`) you pass to the method.

The following example uses these methods to cycle through the data points in a ColumnChart control. It sets the value of the series' `selectedItem` property to identify the new ChartItem as the currently selected item.

```
<?xml version="1.0" ?>
<!--  charts/SimpleCycle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx" result="resultHandler()"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;
        import mx.collections.ArrayCollection;

        [Bindable]
        private var myAC:ArrayCollection = new ArrayCollection();

        private function resultHandler():void {
            myAC = srv.lastResult.data.result;
            series1.selectedIndex = 0;
        }

        private function getNext(e:Event, dir:*):void {
            // Ensure that Flash Player doesn't try to select items outside of the bounds.
            if (series1.selectedIndex < series1.items.length - 1) {
                var curItem:ChartItem = series1.selectedItem;
                var newItem:ChartItem = myChart.getNextItem(dir);
                applyNewItem(newItem);
            }
        }
        private function getPrev(e:Event, dir:*):void {
            // Ensure that Flash Player doesn't try to select items outside of the bounds.
            if (series1.selectedIndex > 0) {
                var curItem:ChartItem = series1.selectedItem;
                var newItem:ChartItem = myChart.getPreviousItem(dir);
                applyNewItem(newItem);
            }
        }
        private function getFirst(e:Event, dir:*):void {
            var newItem:ChartItem = myChart.getFirstItem(dir);
            applyNewItem(newItem);
        }
        private function getLast(e:Event, dir:*):void {
            var newItem:ChartItem = myChart.getLastItem(dir);
            applyNewItem(newItem);
        }
        private function applyNewItem(n:ChartItem):void {
            trace(series1.selectedIndex + " of " + series1.items.length);
            series1.selectedItem = n;
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel height="100%" width="100%">
         <s:layout>
             <s:VerticalLayout/>
         </s:layout>
        <mx:ColumnChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{myAC}"
            selectionMode="single">
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="expenses"
                    displayName="Expenses"
                    selectable="true"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>

        <mx:Legend dataProvider="{myChart}" width="200"/>
        <s:HGroup>
            <s:Button label="|&lt;"
                click="getFirst(event, ChartBase.HORIZONTAL);"/>
            <s:Button label="&lt;"
                click="getPrev(event, ChartBase.HORIZONTAL);"/>
            <s:Button label="&gt;"
                click="getNext(event, ChartBase.HORIZONTAL);"/>
            <s:Button label="&gt;|"
                click="getLast(event, ChartBase.HORIZONTAL);"/>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

These item getter methods do not have associated setters. You cannot set the selected ChartItem objects in the same manner. Instead, you use the properties of the series, as described in “Properties of the series” on page 1363.

Calling the `getNextItem()` and `getPreviousItem()` methods provides more control over the item selection than simply incrementing and decrementing the series’ `selectedIndex` property as shown in “Properties of the series” on page 1363. With these methods, you can choose the direction in which you select the next or previous item, and then decide whether to make the item appear selected.

The item getter methods also account for when you get to the end of the series, for example. If no ChartItems are currently selected, then the `getNextItem()` method gets the first one in the series. If you are at the beginning of the series, the `getPreviousItem()` gets the last item in the series.

The ChartBase class defines two additional properties, `selectedChartItem` and `selectedChartItems`, that return a ChartItem object or an array of ChartItem objects that are selected. You cannot set the values of these properties to select chart items. These properties are read-only. They are useful if your chart has multiple series and you want to know which items across the series are selected without iterating over all the series.

Setting the current state of a ChartItem object

You can set the value of the `currentState` property of a ChartItem object to make it appear selected or deselected, or make it appear in some other state. The `currentState` property can be set to `none`, `rollOver`, `selected`, `disabled`, `focusedSelected`, and `focused`.

Setting the state of the item does not add it to the `selectedItems` array. It only changes the appearance of the chart item. Setting the value of this property also does not trigger a `change` event.

Defined range

You can use the `getItemsInRegion()` method to define a rectangular area and select those chart items that are within that region. The `getItemsInRegion()` method takes an instance of the Rectangle class as its only argument. This rectangle defines an area on the stage, in global coordinates.

Getting an array of ChartItem objects with the `getItemsInRegion()` method does not trigger a `change` event. The state of the items does not change.

The following example lets you specify the `x`, `y`, `height`, and `width` properties of a rectangular range. It then calls the `getItemsInRegion()` method and passes those values to define the range. All chart items that fall within this invisible rectangle are selected and their values are displayed in the TextArea control.

```
<?xml version="1.0" ?>
<!--  charts/GetItemsInRangeExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600" width="600">

    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/medals-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/medals.aspx -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;

        private function getItems(e:Event):void {
            var x:Number = Number(ti1.text);
            var y:Number = Number(ti2.text);
            var h:Number = Number(ti3.text);
            var w:Number = Number(ti4.text);

            var r:Rectangle = new Rectangle(x, y, h, w);

            // Get an Array of ChartItems in the defined area.
            var a:Array = myChart.getItemsInRegion(r);

            for (var i:int = 0; i<a.length; i++) {
                var myChartItem:ColumnSeriesItem = ColumnSeriesItem(a[i]);

                // Make items appear selected.
                myChartItem.currentState = "selected";

                // Show values of the items that appear selected.
                ta1.text += myChartItem.xValue.toString() + 
                    "=" + myChartItem.yValue.toString() + "\n";
            }
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="p1" title="Selecting Items in Ranges">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart"
            height="207" width="350"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}"
            selectionMode="multiple">
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="gold"
                    displayName="Gold"
                    selectable="true"/>
                <mx:ColumnSeries id="series2"
                    yField="silver"
                    displayName="Silver"
                    selectable="true"/>
                <mx:ColumnSeries id="series3"
                    yField="bronze"
                    displayName="bronze"
                    selectable="true"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="country"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}" direction="horizontal"/>

        <s:Label text="Specify dimensions of a rectangle:"/>
        <s:HGroup>
            <s:Form>
                <s:FormItem label="x">
                    <s:TextInput id="ti1" text="20" width="50"/>
                </s:FormItem>
                <s:FormItem label="y">
                    <s:TextInput id="ti2" text="20" width="50"/>
                </s:FormItem>
            </s:Form>
            <s:Form>
                <s:FormItem label="Height">
                    <s:TextInput id="ti3" text="200" width="50"/>
                </s:FormItem>
                <s:FormItem label="Width">
                    <s:TextInput id="ti4" text="200" width="50"/>
                </s:FormItem>
            </s:Form>
        </s:HGroup>
       <s:Button label="Select Items Inside Rectangle" click="ta1.text='';getItems(event)" />
        <s:TextArea id="ta1" height="100" width="300"/>
    </s:Panel>
</s:Application>
```

Working with ChartItem objects to access chart data

To work with the data of a ChartItem, you typically cast it to its specific series type. For example, in a ColumnChart control, the ChartItem objects are of type ColumnSeriesItem. Doing this lets you access series-specific properties such as the `yValue`, `xValue`, and `index` of the ChartItem. You can also then access the data provider's object by using the `item` property. Finally, you can access the properties of the series by casting the `element` property to a series object.

Assuming that `a[i]` is a reference to a ChartItem object from an Array, use the following syntax.

Access the properties of the series item

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
trace(csi.yValue);
```

Access the item's fields

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
trace(csi.item.Profit);
```

Access the series properties

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
var cs:ColumnSeries = ColumnSeries(csi.element);
trace(cs.displayName);
```

About selection events

When a user selects a chart item using mouse, keyboard or region selection, the chart's `change` event is dispatched. When the following properties are updated through user interaction, a `change` event is dispatched:

- `selectedChartItem` and `selectedChartItems` (on the chart)
- `selectedItem`, `selectedItems`, `selectedIndex`, and `selectedIndices` (on the chart's series)

The `change` event does not contain references to the HitData or HitSet of the selected chart items. To access that information, you can use the chart's `selectedChartItem` and `selectedChartItems` properties.

When you select an item programmatically, no `change` event is dispatched. When you change the `selectedState` property of a chart item, no `change` event is dispatched.

Clearing selections

You can clear all selected data points by using the chart control's `clearSelection()` method. The following example clears all selected data points when the user presses the ESC key:

```
<?xml version="1.0" ?>
<!--  charts/ClearItemSelection.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();srv.send();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import flash.events.KeyboardEvent;
        import mx.core.FlexGlobals;

        private function initApp():void {
          FlexGlobals.topLevelApplication.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
        }

        // Clears all chart items selected when the user presses the ESC key.
        private function keyHandler(event:KeyboardEvent):void {
            var curKeyCode:int = event.keyCode;
            if (curKeyCode == 27) { // 27 is the keycode value for ESC
                myChart.clearSelection();
            }
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel height="100%" width="100%">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:PlotChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}"
            selectionMode="multiple">
            <mx:series>
                <mx:PlotSeries id="series1"
                    xField="expenses"
                    yField="profit"
                    displayName="Expenses/Profit"
                    selectable="true"/>
                <mx:PlotSeries id="series2"
                    xField="amount"
                    yField="expenses"
                    displayName="Amount/Expenses"
                    selectable="true"/>
                <mx:PlotSeries id="series3"
                    xField="profit"
                    yField="amount"
                    displayName="Profit/Amount"
                    selectable="true"/>
            </mx:series>
        </mx:PlotChart>
        <mx:Legend dataProvider="{myChart}" width="200"/>
        <s:Label text="Press ESC to clear selection."/>
    </s:Panel>
</s:Application>
```

In addition to clearing item selections programmatically, users can use the mouse or keyboard to clear items. If a user clicks anywhere on the chart control's background, and not over a data point, they clear all selections. If a user uses the up and down arrow keys to select a data point, they clear the existing data points. For more information, see "Keyboard and mouse selection" on page 1362.

Using the selection API to create new charts

You can use the selection API to get some or all of the ChartItem objects of one chart, and then create a new chart with them. To do this, you create a new data provider for the new chart. To do this, you can call the ArrayCollection's `getItemAt()` method on the original chart's data provider and pass to it the original series' selected indices. This method then returns an object whose values you then add to an object in the new data provider.

The `selectedIndex` and `selectedIndices` properties of a chart's series represent the position of the chart item in the series. This position is also equivalent to the position of the chart item's underlying data object in the data provider.

The following example creates a PieChart control from the selected columns in the ColumnChart control. The PieChart control also allows selection; you can explode a piece by selecting it.

```
<?xml version="1.0" ?>
<!--  charts/MakeChartFromSelection.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();initApp();"
    height="600">
    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx" result="resultHandler()"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <mx:SeriesInterpolate id="interpol"
            duration="1000"
            elementOffset="0"
            minimumElementDuration="200"/>
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;
        import mx.charts.PieChart;
        import mx.charts.series.PieSeries;
        import mx.charts.events.ChartItemEvent;
        import mx.charts.Legend;
        [Bindable]
        public var expensesAC:ArrayCollection;
        [Bindable]
        public var newDataProviderAC:ArrayCollection;

        private function resultHandler():void {
               expensesAC = ArrayCollection(srv.lastResult.data.result);
        }

        private function initApp():void {
            myColumnChart.addEventListener(ChartItemEvent.CHANGE, createNewChart);
            setupPieChart();
        }
        private function getNewDataProvider():ArrayCollection {
            newDataProviderAC = new ArrayCollection();

            for (var i:int=0; i<series1.selectedItems.length; i++) {
                var o:Object = new Object();
                o.Month = expensesAC.getItemAt(series1.selectedIndices[i]).month;
                o.Expenses = expensesAC.getItemAt(series1.selectedIndices[i]).expenses;
                newDataProviderAC.addItem(o);
            }
            return newDataProviderAC;
        }
        private var newChart:PieChart;
        private var newSeries:PieSeries;

        [Bindable]
        private var explodedPiece:Array;
        private function explodePiece(e:Event):void {
            explodedPiece = new Array();
            explodedPiece[newSeries.selectedIndex] = .2;
            newSeries.perWedgeExplodeRadius = explodedPiece;
        }
        private function setupPieChart():void {
            newChart  = new PieChart();
            newChart.showDataTips = true;
            newChart.selectionMode = "single";
            newSeries = new PieSeries();
            newSeries.field = "Expenses";
            newSeries.nameField = "Month";
            newSeries.setStyle("labelPosition", "callout");
            newSeries.setStyle("showDataEffect", "interpol");
            var newSeriesArray:Array = new Array();
            newSeriesArray.push(newSeries);
            newChart.series = newSeriesArray;
            newChart.addEventListener(ChartItemEvent.CHANGE, explodePiece);
            // Create a legend for the new chart.
            var newLegend:Legend = new Legend();
            newLegend.dataProvider = newChart;
            p1.addElement(newChart);
            p1.addElement(newLegend);
        }
        private function createNewChart(e:Event):void {
            newChart.dataProvider = getNewDataProvider();
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="p1" title="Column Chart">
        <s:layout>
```

```
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myColumnChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}"
            selectionMode="multiple">
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="expenses"
                    displayName="Expenses"
                    selectable="true"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
    </s:Panel>
</s:Application>
```

Dragging and dropping ChartItem objects

As an extension of the selection API, you can drag and drop chart items from one chart to another (or from a chart to some other object altogether).

To use drag and drop operations in your chart controls:

- Make the source chart drag enabled by setting it's `dragEnabled` property to `true`.
- Make the target chart drop enabled by setting it's `dropEnabled` property to `true`.
- Add event listeners for the `dragEnter` and `dragDrop` events on the target chart.
- In the `dragEnter` event handler, get a reference to the target chart and register it with the DragManager. To be a drop target, a chart must define an event handler for this event. You must call the `DragManager.acceptDragDrop()` method for the drop target to receive the drag and drop events.
- In the `dragDrop` event handler, get an Array of chart items that are being dragged and add that Array to the target chart's data provider. You do this by using the `event.dragSource.dataForFormat()` method. You must specify the String `"chartitems"` as the argument to the `dataForFormat()` method. This is because the chart-based controls have predefined values for the data format of drag data. For all chart controls, the format String is `"chartitems"`.

The following example lets you drag chart items from the ColumnChart control to the PieChart control. When the application starts, there is no PieChart control, but it becomes visible when the first chart item is dragged onto the container. This example also examines the dragged chart items to ensure that they are not added more than once to the target chart. It does this by using the target data provider's `contains()` method.

```
<?xml version="1.0" ?>
<!--  charts/MakeChartFromDragDrop.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();initApp();"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx" result="resultHandler()"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <mx:SeriesInterpolate id="interpol"
            duration="1000"
            elementOffset="0"
            minimumElementDuration="200"
        />
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.PieChart;
        import mx.charts.series.PieSeries;
        import mx.charts.events.ChartItemEvent;
        import mx.events.DragEvent;
        import mx.controls.List;
        import mx.managers.DragManager;
        import mx.core.DragSource;
        import mx.charts.Legend;
        [Bindable]
        public var newDataProviderAC:ArrayCollection;

        [Bindable]
        private var expensesAC:ArrayCollection;

        private function initApp():void {
            setupPieChart();
        }
        private function resultHandler():void {
               expensesAC = ArrayCollection(srv.lastResult.data.result);
        }

        private var newChart:PieChart;
        private var newSeries:PieSeries;

        [Bindable]
        private var explodedPiece:Array;
        private function explodePiece(e:Event):void {
            explodedPiece = new Array();
            explodedPiece[newSeries.selectedIndex] = .2;
            newSeries.perWedgeExplodeRadius = explodedPiece;
        }
        private function setupPieChart():void {
            newChart  = new PieChart();
            newChart.showDataTips = true;
            newChart.selectionMode = "multiple";
            newChart.dropEnabled= true;
            newChart.dragEnabled= false;

            newChart.height = 350;
            newChart.width = 350;

            newChart.addEventListener("dragEnter", doDragEnter);
            newChart.addEventListener("dragDrop", doDragDrop);
            newChart.dataProvider = newDataProviderAC;
            newSeries = new PieSeries();
            newSeries.field = "expenses";
            newSeries.nameField = "month";
            newSeries.setStyle("labelPosition", "callout");
            newSeries.setStyle("showDataEffect", "interpol");
            var newSeriesArray:Array = new Array();
            newSeriesArray.push(newSeries);
            newChart.series = newSeriesArray;
            newChart.addEventListener(ChartItemEvent.CHANGE, explodePiece);
            // Create a legend for the new chart.
            var newLegend:Legend = new Legend();
            newLegend.dataProvider = newChart;
            p2.addElement(newChart);
            p2.addElement(newLegend);
        }
        private function doDragEnter(event:DragEvent):void {
            // Get a reference to the target chart.
            var dragTarget:ChartBase = ChartBase(event.currentTarget);
            // Register the target chart with the DragManager.
            DragManager.acceptDragDrop(dragTarget);
        }
        private function doDragDrop(event:DragEvent):void {
            // Get a reference to the target chart.
            var dropTarget:ChartBase=ChartBase(event.currentTarget);
            // Get the dragged items from the drag initiator. When getting
            // items from chart controls, you must use the 'chartitems'
            // format.
            var items:Array = event.dragSource.dataForFormat("chartitems")
                as Array;
            // Trace status messages.
            trace("length: " + String(items.length));
            trace("format: " + String(event.dragSource.formats[0]));

            // Add each item to the drop target's data provider.
            for(var i:uint=0; i < items.length; i++) {

                // If the target data provider already contains the
                // item, then do nothing.
                if (dropTarget.dataProvider.contains(items[i].item)) {

                // If the target data provider does NOT already
                // contain the item, then add it.
                } else {
                    dropTarget.dataProvider.addItem(items[i].item);
                }
            }
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="p1" title="Source Chart" height="250" width="400">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myColumnChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}"
            selectionMode="multiple"
            dragEnabled="true"
            dropEnabled="false">
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="expenses"
                    displayName="Expenses"
                    selectable="true"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
    </s:Panel>

    <!-- This will be the parent of the soon-to-be created chart. -->
    <s:Panel id="p2" title="Target Chart" height="400" width="400">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Panel>
</s:Application>
```

Chart controls also support the standard drag and drop methods of List-based controls such as `hideDropFeedback()` and `showDropFeedback()`. These methods display indicators under the mouse pointer to indicate whether drag and drop operations are allowed and where the items will be dropped.

For more information about drag and drop operations, see “Drag and drop” on page 1893.

Dropping ChartItem objects onto components

The target of a drag and drop operation from a chart control does not need to be another chart control. Instead, you can drag an object onto any Flex component using simple drag and drop rules. The following example lets you drag a column from the chart onto the TextArea. The TextArea then extracts data from the dropped item and displays that information in text.

```
<?xml version="1.0" ?>
<!--  charts/DragDropToComponent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/medals-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/medals.aspx -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.events.DragEvent;
        import mx.controls.List;
        import mx.managers.DragManager;
        import mx.core.DragSource;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.events.ChartItemEvent;
        import mx.charts.series.items.ColumnSeriesItem;
        private function doDragEnter(event:DragEvent):void {
            var dragTarget:TextArea = TextArea(event.currentTarget);
            DragManager.acceptDragDrop(dragTarget);
        }
        private function doDragDrop(event:DragEvent):void {
            var dropTarget:TextArea = TextArea(event.currentTarget);

            var curItem:ColumnSeriesItem =
               ColumnSeriesItem(event.dragSource.dataForFormat("chartitems")[0]);
            var curSeries:ColumnSeries = ColumnSeries(curItem.element);

            var medalType:String = curSeries.displayName;
            var numMedals:String = curItem.yValue.toString();
            var countryName:String = curItem.item.country;

            ta1.text = countryName + " earned " + numMedals + " " + medalType + " medals.";
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Dropping ChartItem Objects">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart"
            height="225"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}"
            selectionMode="single"
            dragEnabled="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="country"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries id="columnSeries1"
                    xField="country"
                    yField="gold"
                    displayName="Gold"
                    selectable="true"/>
                <mx:ColumnSeries id="columnSeries2"
                    xField="country"
                    yField="silver"
                    displayName="Silver"
                    selectable="true"/>
                <mx:ColumnSeries id="columnSeries3"
                    xField="country"
                    yField="bronze"
                    displayName="Bronze"
                    selectable="true"/>
            </mx:series>
        </mx:ColumnChart>
        <s:HGroup>
            <mx:Legend dataProvider="{myChart}"/>
            <s:TextArea id="ta1"
                height="75" width="200"
                dragEnter="doDragEnter(event)"
                dragDrop="doDragDrop(event)"/>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

Changing the drag image

When you drag a chart item off of a source chart, an outline of the underlyig chart item appears as the drag image. For example, if you drag a column from a ColumnChart control, a column appears under the mouse pointer to represent the item that is being dragged.

You can customize the image that is displayed during a drag operation by overriding the default defintion of the `dragImage` property in a custom chart class. You do this by embedding your new image (or defining it in ActionScript), overriding the `dragImage()` getter method, and returning the new image.

The default location, in coordinates, of the drag image is 0,0 unless you override the DragManager's `doDrag()` method. You can also set the starting location of the drag proxy image by using the x and y coordinates of the image proxy in the `dragImage()` getter.

The following example custom chart class embeds an image and returns it in the `dragImage()` getter method. This example also positions the drag image proxy so that the mouse pointer is near its lower right corner.

```
// charts/MyColumnChart.as
package {
    import mx.charts.ColumnChart;
    import mx.core.IUIComponent;
    import mx.controls.Image;
    public class MyColumnChart extends ColumnChart {
        [Embed(source="images/dollarSign.png")]
        public var dollarSign:Class;
        public function MyColumnChart() {
            super();
        }

        override protected function get dragImage():IUIComponent {
             var imageProxy:Image = new Image();
             imageProxy.source = dollarSign;

             var imageHeight:Number = 50;
             var imageWidth:Number = 32;

             imageProxy.height = imageHeight;
             imageProxy.width = imageWidth;
             // Position the image proxy above and to the left of
             // the mouse pointer.
             imageProxy.x = this.mouseX - imageWidth;
             imageProxy.y = this.mouseY - imageHeight;
             return imageProxy;
        }

    }
}
```

The following example uses the MyColumnChart example class to define its custom drag image:

```
<?xml version="1.0" ?>
<!--  charts/DragDropCustomDragImage.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send();initApp();"
    xmlns:local="*"
    height="800">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
        <mx:SeriesInterpolate id="interpol"
            duration="1000"
            elementOffset="0"
            minimumElementDuration="200"/>
    </fx:Declarations>

    <fx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.PieChart;
        import mx.charts.series.PieSeries;
        import mx.charts.events.ChartItemEvent;
        import mx.events.DragEvent;
        import mx.controls.List;
        import mx.managers.DragManager;
        import mx.core.DragSource;
        import mx.controls.Image;
        import flash.events.MouseEvent;
        import mx.charts.Legend;
        [Bindable]
        public var newDataProviderAC:ArrayCollection;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection([
            { Month: "Jan", Expenses: 1500 },
            { Month: "Feb", Expenses: 200 },
            { Month: "Mar", Expenses: 500 },
            { Month: "Apr", Expenses: 1200 },
            { Month: "May", Expenses: 575 } ]);

        private function initApp():void {
            setupPieChart();
        }

        private var newChart:PieChart;
        private var newSeries:PieSeries;

        [Bindable]
        private var explodedPiece:Array;
        private function explodePiece(e:Event):void {
            explodedPiece = new Array();
            explodedPiece[newSeries.selectedIndex] = .2;
            newSeries.perWedgeExplodeRadius = explodedPiece;
        }
        private function setupPieChart():void {
            newChart  = new PieChart();
            newChart.showDataTips = true;
            newChart.selectionMode = "multiple";
            newChart.dropEnabled= true;
            newChart.dragEnabled= false;
            newChart.height = 350;
            newChart.width = 350;

            newChart.addEventListener("dragEnter", doDragEnter);
            newChart.addEventListener("dragDrop", doDragDrop);
            newChart.dataProvider = newDataProviderAC;
            newSeries = new PieSeries();
            newSeries.field = "expenses";
            newSeries.nameField = "month";
            newSeries.setStyle("labelPosition", "callout");
            newSeries.setStyle("showDataEffect", "interpol");
            var newSeriesArray:Array = new Array();
            newSeriesArray.push(newSeries);
            newChart.series = newSeriesArray;
            newChart.addEventListener(ChartItemEvent.CHANGE, explodePiece);
            // Create a legend for the new chart.
            var newLegend:Legend = new Legend();
            newLegend.dataProvider = newChart;
            p2.addElement(newChart);
            p2.addElement(newLegend);
        }
        private function doDragEnter(event:DragEvent):void {
            var dragTarget:ChartBase = ChartBase(event.currentTarget);
            DragManager.acceptDragDrop(dragTarget);
        }
        private function doDragDrop(event:DragEvent):void {
            var dropTarget:ChartBase=ChartBase(event.currentTarget);
            var items:Array = event.dragSource.dataForFormat("chartitems") as Array;
            for(var i:uint=0; i < items.length; i++) {
                if (dropTarget.dataProvider.contains(items[i].item)) {
                } else {
                    dropTarget.dataProvider.addItem(items[i].item);
                }
            }
        }
    ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="p1" title="Source Chart"
        height="250" width="400">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <local:MyColumnChart id="myChart"
            height="207" width="350"
            showDataTips="true"
            dataProvider="{srv.lastResult.data.result}"
            selectionMode="multiple"
            dragEnabled="true"
            dropEnabled="false">
            <local:series>
                <mx:ColumnSeries id="series1"
                    yField="expenses"
                    displayName="Expenses"
                    selectable="true"/>
            </local:series>
            <local:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </local:horizontalAxis>
        </local:MyColumnChart>
    </s:Panel>

    <s:Panel id="p2" title="Target Chart"
        height="400" width="400">
    </s:Panel>
</s:Application>
```

Drawing on chart controls

Flex includes the ability to add graphical elements to your chart controls such as lines, boxes, and elipses. Graphical children of chart controls can also include any control that is a subclass of UIComponent, including list boxes, combo boxes, labels, and even other chart controls.

When working with chart controls, Flex converts x and y coordinates into data coordinates. Data coordinates define locations relative to the data in the chart's underlying data provider. This lets you position graphical elements relative to the location of data points in the chart without having to first convert their positions to x and y coordinates. For example, you can draw a line from the top of a column in a ColumnChart to the the top of another column, showing a trend line.

To add data graphics to a chart control, you use the mx.charts.chartClasses.CartesianDataCanvas (for Cartesian charts) class or the mx.charts.chartClasses.PolarDataCanvas (for polar charts) classes. You can either attach visual controls to the canvas or you can draw on the canvas by using drawing methods.

You attach visual controls to the data canvases in the same way that you programmatically add controls to an application: you add them as child controls. In this case, the method you call to add children is `addDataChild()`, which lets you add any DisplayObject instance to the canvas. This method lets you take advantage of the data coordinates that the canvas uses. As with most MX containers, you can also use the `addChild()` and `addChildAt()` methods. With these methods, you can then adjust the location of the DisplayObject to data coordinates by using the canvas' `updateDataChild()` method.

To draw on the data canvases, you use drawing methods that are similar to those used in MXML graphics. The canvases define a set of methods that you can use to create vector shapes such as circles, squares, lines, and other shapes. These methods include the line-drawing methods such as `lineTo()`, `moveTo()`, and `curveTo()`, as well as the fill methods such as `beginFill()`, `beginBitmapFill()`, and `endFill()`. Convenience methods such as `drawRect()`, `drawRoundedRect()`, `drawEllipse()`, and `drawCircle()` are also available on the data canvases. For more information about MXML graphics, see "MXML graphics" on page 1730.

All drawn graphics such as lines and shapes remain visible on the data canvas until you call the canvas' `clear()` method. To remove child objects from data canvases, use the `removeChild()`, `removeChildAt()`, and `removeAllChildren()` methods.

A canvas can either be in the foreground (in front of the data points) or in the background (behind the data points). To add a canvas to the foreground, you add it to the chart's `annotationElement` Array. To add a canvas to the background, you add it to the chart's `backgroundElements` Array.

The data canvases have the following limitations:

- There is no drag-and-drop support for children of the data canvases.
- You cannot use the chart selection APIs to select objects on the data canvases.

The following example adds a CartesianDataCanvas as an annotation element to the ColumnChart control. It uses the graphics methods to draw a line between the columns that you select.

```
<?xml version="1.0"?>
<!-- charts/DrawLineBetweenSelectedItems.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
  <fx:Script><![CDATA[
    import mx.charts.series.items.ColumnSeriesItem;
    import mx.charts.ChartItem;
     private function connectTwoPoints(month1:String, 
        value1:Number, 
        month2:String, 
        value2:Number
     ):void {
        canvas.clear();
        canvas.lineStyle(4, 
            0xCCCCCC, 
            .75, 
            true, 
            LineScaleMode.NORMAL, 
            CapsStyle.ROUND, 
            JointStyle.MITER, 
            2
        );
        canvas.moveTo(month1, value1);
        canvas.lineTo(month2, value2);

        l1.text = "Month: " + month1;
        l2.text = "Expense: " + value1;
        l3.text = "Month: " + month2;
        l4.text = "Expense: " + value2;
        chartHasLine = true;
     }
    private var s1:String = new String();
    private var s2:String = new String();
    private var v1:Number = new Number();
    private var v2:Number = new Number();
    // Set this to true initially so that the chart does not
    // draw a line when the first item is clicked.
    private var chartHasLine:Boolean = true;
    private function handleChange(event:Event):void {
        var sci:ColumnSeriesItem = 
            ColumnSeriesItem(myChart.selectedChartItem);
        if (chartHasLine) {
            canvas.clear();
            s1 = sci.item.month;
            v1 = sci.item.expenses;
            chartHasLine = false;
        } else {
            s2 = sci.item.month;
            v2 = sci.item.expenses;
            connectTwoPoints(s1, v1, s2, v2);
        }
    }
]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
<s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        dataProvider="{srv.lastResult.data.result}"
        selectionMode="single"
        change="handleChange(event)">
        <mx:annotationElements>
            <mx:CartesianDataCanvas id="canvas" includeInRanges="true"/>
        </mx:annotationElements>
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                id="series1"
                xField="month"
                yField="expenses"
                displayName="Expense"
                selectable="true"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

     <s:HGroup>
        <s:Button id="b1"
            label="Connect Two Points"
            click="connectTwoPoints('Jan', 1500, 'Mar', 500);"/>
        <s:Button id="b2"
            click="canvas.clear()"
            label="Clear Line"/>
     </s:HGroup>

     <s:HGroup>
         <s:VGroup>
             <s:Label text="First Item"/>
             <s:Label id="l1"/>
             <s:Label id="l2"/>
         </s:VGroup>
         <s:VGroup>
             <s:Label text="Second Item"/>
             <s:Label id="l3"/>
             <s:Label id="l4"/>
         </s:VGroup>
     </s:HGroup>
   </s:Panel>
</s:Application>
```

The following example uses the `addDataChild()` method to add children to the data canvas. It adds labels to each of the columns that you select in the ColumnChart control.

```
<?xml version="1.0"?>
<!-- charts/AddLabelsWithLines.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script><![CDATA[
    import mx.charts.series.items.ColumnSeriesItem;
    import mx.charts.ChartItem;
     private function connectTwoPoints(month1:String,
        value1:Number,
        month2:String,
        value2:Number
     ):void {
        canvas.clear();
        canvas.lineStyle(4,
            0xCCCCCC,
            .75,
            true,
            LineScaleMode.NORMAL,
            CapsStyle.ROUND,
            JointStyle.MITER,
            2
        );
        canvas.moveTo(month1, value1);
        canvas.lineTo(month2, value2);

        l1.text = "Month: " + month1;
        l2.text = "Profit: " + value1;
        l3.text = "Month: " + month2;
        l4.text = "Profit: " + value2;
        chartHasLine = true;
     }
    private var s1:String = new String();
    private var s2:String = new String();
    private var v1:Number = new Number();
    private var v2:Number = new Number();
    // Set this to true initially so that the chart doesn't
    // draw a line when the first item is clicked.
    private var chartHasLine:Boolean = true;
    private function handleChange(event:Event):void {
        var sci:ColumnSeriesItem =
            ColumnSeriesItem(myChart.selectedChartItem);
        if (chartHasLine) {
            canvas.clear();
            s1 = sci.item.month;
            v1 = sci.item.profit;
            addLabelsToColumn(s1,v1);

            chartHasLine = false;
        } else {
            s2 = sci.item.month;
            v2 = sci.item.profit;
            addLabelsToColumn(s2,v2);

            connectTwoPoints(s1, v1, s2, v2);
        }
    }
    [Bindable]
    public var columnLabel:Label;

    private function addLabelsToColumn(s:String, n:Number):void {
        columnLabel = new Label();
        columnLabel.setStyle("fontWeight", "bold");
        columnLabel.setStyle("color", "0x660000");
        columnLabel.text = s + ": " + "$" + n;

        // This adds any DisplayObject as child to current canvas.
        canvas.addDataChild(columnLabel, s, n);
    }
  ]]></fx:Script>
  <s:Panel title="Column Chart">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:ColumnChart id="myChart"
```

```
        dataProvider="{srv.lastResult.data.result}"
        selectionMode="single"
        change="handleChange(event)">
        <mx:annotationElements>
            <mx:CartesianDataCanvas id="canvas" includeInRanges="true"/>
        </mx:annotationElements>
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                id="series1"
                xField="month"
                yField="profit"
                displayName="Profit"
                selectable="true"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

     <s:Button id="b1" label="Connect Two Points"
        click="connectTwoPoints('Jan', 2000, 'Mar', 1500);"/>

     <s:HGroup>
         <s:VGroup>
             <s:Label text="First Item"/>
             <s:Label id="l1"/>
             <s:Label id="l2"/>
         </s:VGroup>
         <s:VGroup>
             <s:Label text="Second Item"/>
             <s:Label id="l3"/>
             <s:Label id="l4"/>
         </s:VGroup>
     </s:HGroup>

  </s:Panel>
</s:Application>
```

You can access an array of the data children by using the `dataChildren` property of the canvas. This property is an array of the child objects on the canvas.

The following example uses the `updateDataChild()` method to add a label to a single data point on the line:

```
<?xml version="1.0"?>
<!-- charts/UpdateDataChildExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
  <fx:Script><![CDATA[
    import mx.controls.Label;
    import mx.charts.chartClasses.CartesianCanvasValue;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);

     public function drawData():void
     {
        canvas.clear();
        canvas.beginFill(0xFF0033, 1);
        canvas.drawCircle("Feb", 1000, 20);
        canvas.endFill();

        var myLabel:Label = new Label();
        myLabel.text = "X";
        myLabel.setStyle("fontWeight", "bold");
        myLabel.setStyle("fontSize", 20);
        canvas.addChild(myLabel);
        canvas.updateDataChild(myLabel, "Feb", 1100);
     }

  ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Panel title="Line Chart">
     <s:layout>
         <s:HorizontalLayout/>
     </s:layout>
     <mx:LineChart id="myChart" dataProvider="{expenses}"
        showDataTips="true" creationComplete="drawData()">
        <mx:annotationElements>
            <mx:CartesianDataCanvas id="canvas" includeInRanges="true" />
        </mx:annotationElements>
        <mx:horizontalAxis>
           <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="Profit" displayName="Profit"/>
           <mx:LineSeries yField="Expenses" displayName="Expenses"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </s:Panel>
</s:Application>
```

Using offsets for data coordinates

The data canvas classes also let you add offsets to the position of data graphics. You do this by defining the data coordinates with the CartesianCanvasValue constructor rather than passing a data coordinate to the drawing or `addDataChild()` methods. When you define a data coordinate with the CartesianCanvasValue, you pass the data coordinate as the first argument, but you can pass an offset as the second argument.

The following example lets you specify an offset for the labels with an HSlider control. This offset is used when adding a data child to the canvas.

```
<?xml version="1.0"?>
<!-- charts/AddLabelsWithOffsetLines.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">

    <fx:Declarations>
       <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
        <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Script><![CDATA[
    import mx.charts.series.items.ColumnSeriesItem;
    import mx.charts.ChartItem;
    import mx.charts.chartClasses.CartesianCanvasValue;

     private function connectTwoPoints(month1:String,
        value1:Number,
        month2:String,
        value2:Number
     ):void {
        canvas.clear();
        canvas.lineStyle(4,
            0xCCCCCC,
            .75,
            true,
            LineScaleMode.NORMAL,
            CapsStyle.ROUND,
            JointStyle.MITER,
            2
        );
        canvas.moveTo(month1, value1);
        canvas.lineTo(month2, value2);

        l1.text = "Month: " + month1;
        l2.text = "Profit: " + value1;
```

```
      l3.text = "Month: " + month2;
      l4.text = "Profit: " + value2;
      chartHasLine = true;
   }
  private var s1:String = new String();
  private var s2:String = new String();
  private var v1:Number = new Number();
  private var v2:Number = new Number();
  // Set this to true initially so that the chart doesn't
  // draw a line when the first item is clicked.
  private var chartHasLine:Boolean = true;
  private function handleChange(event:Event):void {
      var sci:ColumnSeriesItem =
          ColumnSeriesItem(myChart.selectedChartItem);
      if (chartHasLine) {
          canvas.clear();
          s1 = sci.item.month;
          v1 = sci.item.profit;
          addLabelsToColumn(s1,v1);

          chartHasLine = false;
      } else {
          s2 = sci.item.month;
          v2 = sci.item.profit;
          addLabelsToColumn(s2,v2);

          connectTwoPoints(s1, v1, s2, v2);
      }
  }
  [Bindable]
  public var labelOffset:Number = 0;
  [Bindable]
  public var columnLabel:Label;

  private function addLabelsToColumn(s:String, n:Number):void {
      columnLabel = new Label();
      columnLabel.setStyle("fontWeight", "bold");
      columnLabel.setStyle("color", "0x660000");
      columnLabel.text = s + ": " + "$" + n;

      // Use the CartesianCanvasValue constructor to specify
      // an offset for data coordinates.
      canvas.addDataChild(columnLabel,
          new CartesianCanvasValue(s, labelOffset),
          new CartesianCanvasValue(n, labelOffset)
      );
  }
]]></fx:Script>
<s:Panel title="Column Chart">
      <s:layout>
          <s:VerticalLayout/>
      </s:layout>
   <mx:ColumnChart id="myChart"
      dataProvider="{srv.lastResult.data.result}"
      selectionMode="single"
      change="handleChange(event)">
      <mx:annotationElements>
```

```
            <mx:CartesianDataCanvas id="canvas" includeInRanges="true"/>
        </mx:annotationElements>
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                id="series1"
                xField="month"
                yField="profit"
                displayName="Profit"
                selectable="true"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

     <s:Button id="b1"
        label="Connect Two Points"
        click="connectTwoPoints('Jan', 2000, 'Mar', 1500);"/>

     <s:HGroup>
         <s:VGroup>
             <s:Label text="First Item"/>
             <s:Label id="l1"/>
             <s:Label id="l2"/>
         </s:VGroup>
         <s:VGroup>
             <s:Label text="Second Item"/>
             <s:Label id="l3"/>
             <s:Label id="l4"/>
         </s:VGroup>
     </s:HGroup>

    <mx:HSlider id="hSlider" minimum="-50" maximum="50" value="0"
        dataTipPlacement="top"
        tickColor="black"
        snapInterval="1" tickInterval="10"
        labels="['-50','0','50']"
        allowTrackClick="true"
        liveDragging="true"
        change="labelOffset=hSlider.value"/>
  </s:Panel>
</s:Application>
```

Using multiple axes with data canvases

A CartesianDataCanvas is specific to a certain data space, which is defined by the bounds of the axis. If no axis is specified, the canvas uses the primary axes of the chart as its bounds. If you have a chart with multiple axes, you can specify which axes the data canvas should use by setting the values of the `horizontalAxis` or `verticalAxis` properties, as the following example illustrates:

```
<mx:ColumnChart width="100%" height="100%" creationComplete="createLabel();">
    <mx:annotationElements>
        <mx:CartesianDataCanvas id="canvas"
            includeInRanges="true"
            horizontalAxis={h1}
            verticalAxis={v1}/>
    </mx:annotationElements>
    ...
</mx:ColumnChart>
```

AdvancedDataGrid control

The AdvancedDataGrid control expands on the functionality of the standard MX DataGrid control to add data visualization capabilities to your applications built in Adobe® Flex™. These capabilities provide greater control of data display, data aggregation, and data formatting.

For more information on the DataGrid control, see "MX DataGrid control" on page 963.

There is also a Spark DataGrid control. For more information, see "Spark DataGrid and Grid controls" on page 545.

About the AdvancedDataGrid control

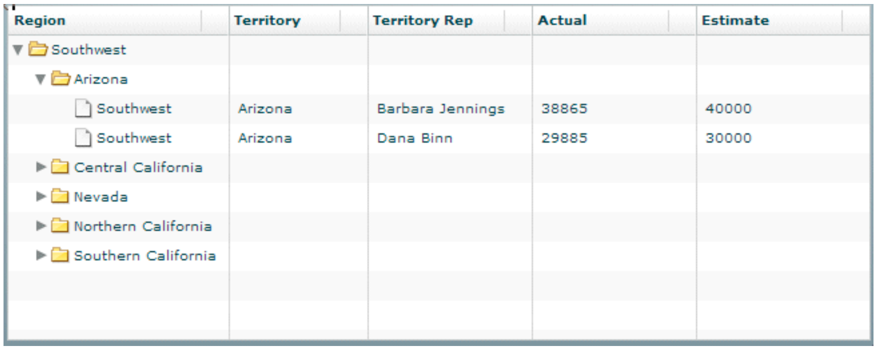
The AdvancedDataGrid control extends the capabilities of the standard DataGrid control to improve data visualization. The following table describes the main data visualization capabilities of the AdvancedDataGrid control:

Capability	**Description**
Sorting by multiple columns	Sort by multiple columns when you click in the column header. For more information, see "Sorting by multiple columns" on page 1399.
Styling rows and columns	Use the `styleFunction` property to specify a function to apply styles to rows and columns of the control. For more information, see "Styling rows and columns" on page 1401.
Displaying hierarchical and grouped data	Use an expandable navigation tree in a column to control the visible rows of the control. For more information, see "Hierarchical and grouped data display" on page 1409.
Creating column groups	Collect multiple columns under a single column heading. For more information, see "Creating column groups" on page 1436.
Using item renderers	Span multiple columns with an item renderer and use multiple item renderers in the same column. For more information, see "Using item renderers with the AdvancedDataGrid control" on page 1442.

Displaying hierarchical and grouped data

One of the most important aspects of the AdvancedDataGrid control is its support for the display of hierarchical and grouped data. *Hierarchical data* is data already in a structure of parent and child data items. *Grouped data* is flat data with no inherent hierarchy. Before passing flat data to the AdvancedDataGrid control, you specify one or more data fields that are used to group the flat data into a hierarchy.

To support the display of hierarchical and grouped data, the AdvancedDataGrid control displays a navigation tree in a column that lets you navigate the data hierarchy. The following example shows an AdvancedDataGrid control with a navigation tree in the first column that controls the visible rows of the control, but it can be in any column:



Region	Territory	Territory Rep	Actual	Estimate
▼ Southwest				
▼ Arizona				
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
▶ Central California				
▶ Nevada				
▶ Northern California				
▶ Southern California				

Migrating from the DataGrid control

Besides the major new features added to the AdvancedDataGrid control described in the section "About the AdvancedDataGrid control" on page 1397, the AdvancedDataGrid control adds the following property:

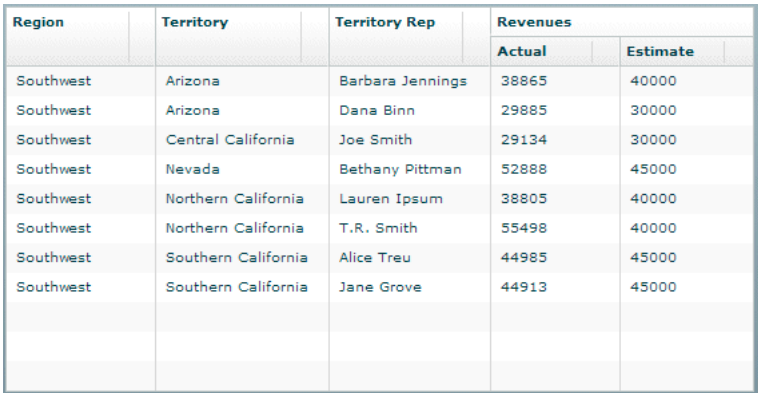
firstVisibleItem Specifies the item that is currently displayed in the top row of the AdvancedDataGrid control.

One consideration when migrating an existing application from the DataGrid control to the AdvancedDataGrid control is that the data type of many event objects dispatched by the AdvancedDataGrid control is of type AdvancedDataGridEvent, not of type DataGridEvent as it is for the DataGrid control. If your application references the event object by type, you must update your application so that it uses the correct data type.

To determine the type of the event object for any event, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Example: Creating column groups

Column grouping lets you collect multiple columns under a single column heading. The following example shows the Actual and Estimate columns grouped under the Revenues column:



Region	Territory	Territory Rep	Revenues	
			Actual	Estimate
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
Southwest	Central California	Joe Smith	29134	30000
Southwest	Nevada	Bethany Pittman	52888	45000
Southwest	Northern California	Lauren Ipsum	38805	40000
Southwest	Northern California	T.R. Smith	55498	40000
Southwest	Southern California	Alice Treu	44985	45000
Southwest	Southern California	Jane Grove	44913	45000

The AdvancedDataGrid control is defined by the mx.controls.AdvancedDataGrid class and additional classes in the package mx.controls.advancedDataGridClasses. This package includes the AdvancedDataGridColumn class that you use to define a column.

The following code creates the column grouping shown in the previous image:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupADG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Import the data used by the AdvancedDataGrid control.
            include "SimpleFlatData.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</s:Application>
```

This example imports the data used by the AdvancedDataGrid control from a file named SimpleFlatData.as. You can see the contents of that file in the topic “Hierarchical and grouped data display” on page 1409.

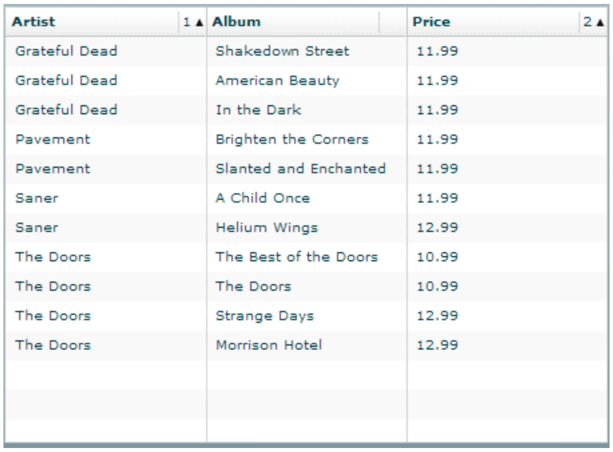
Sorting by multiple columns

By default, the AdvancedDataGrid control displays data in the order specified in the data passed to its `dataProvider` property. The AdvancedDataGrid control also lets you sort data after the control displays it in a single column or multiple columns.

To disable sorting for an entire AdvancedDataGrid control, set the `AdvancedDataGrid.sortableColumns` property to `false`. To disable sorting for an individual column, set the `AdvancedDataGridColumn.sortable` property to `false`.

The way that you sort multiple columns is based on the setting of the `sortExpertMode` property. By default, the `sortExpertMode` property is set to `false`. This setting means that you click in the header area of a column to sort the rows of the AdvancedDataGrid control by that column. Then you click in the multiple column sort area of the header to sort by additional columns. To use the Control key to select every column after the first column to perform sort, set the `sortExpertMode` property to `true`.

The following example shows an AdvancedDataGrid control with three columns with the `sortExpertMode` property set to `false`:



Artist 1 ▲	Album	Price 2 ▲
Grateful Dead	Shakedown Street	11.99
Grateful Dead	American Beauty	11.99
Grateful Dead	In the Dark	11.99
Pavement	Brighten the Corners	11.99
Pavement	Slanted and Enchanted	11.99
Saner	A Child Once	11.99
Saner	Helium Wings	12.99
The Doors	The Best of the Doors	10.99
The Doors	The Doors	10.99
The Doors	Strange Days	12.99
The Doors	Morrison Hotel	12.99

A. *Column header area* ***B*** *. Multiple column sort area*

Sort the columns with sortExpertMode set to false

1. Click in the column header area of any column in the AdvancedDataGrid control to sort by that column. For example, click in the column header for the Artist column to sort that column in ascending order. Click the Artist column header again to sort in descending order.
2. Click in the multiple column sort area of any other column header. For example, click in the multiple column sort area of the Price column to arrange it in ascending order while keeping the Artist column sorted in descending order. You can now find the least expensive album for each artist.
3. Click in the multiple column sort area for the Price column again to arrange it in descending order.
4. Click in the multiple column sort area for any other column header to include other columns in the sort.

Sort the columns with sortExpertMode set to true

1. Click in the column header area of any column in the AdvancedDataGrid control to sort by that column. For example, click in the column header for the Artist column to sort that column in ascending order. Click the Artist column header again to sort in descending order.
2. While holding down the Control key, click in any other column header. For example, click the column header for the Price column to arrange it in ascending order while keeping the Artist column sorted in descending order. You can now find the least expensive album for each artist.
3. While holding down the Control key, click the column header for the Price column again to arrange it in descending order.
4. While holding down the Control key, select any other column header to include other columns in the sort.

The following code implements multiple column sort when the `sortExpertMode` property is set to `true`:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleADG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
      <![CDATA[
         import mx.collections.ArrayCollection;

         [Bindable]
         private var dpADG:ArrayCollection = new ArrayCollection([
           {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
           {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99},
           {Artist:'Saner', Album:'A Child Once', Price:11.99},
           {Artist:'Saner', Album:'Helium Wings', Price:12.99},
           {Artist:'The Doors', Album:'The Doors', Price:10.99},
           {Artist:'The Doors', Album:'Morrison Hotel', Price:12.99},
           {Artist:'Grateful Dead', Album:'American Beauty', Price:11.99},
           {Artist:'Grateful Dead', Album:'In the Dark', Price:11.99},
           {Artist:'Grateful Dead', Album:'Shakedown Street', Price:11.99},
           {Artist:'The Doors', Album:'Strange Days', Price:12.99},
           {Artist:'The Doors', Album:'The Best of the Doors', Price:10.99}
         ]);
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid
        width="100%" height="100%"
        sortExpertMode="true"
        dataProvider="{dpADG}">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist" />
            <mx:AdvancedDataGridColumn dataField="Album" />
            <mx:AdvancedDataGridColumn dataField="Price" />
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

Styling rows and columns

You apply styles to the rows and columns of the AdvancedDataGrid control by using callback functions. To control the styling of a row, use the `AdvancedDataGrid.styleFunction` property to specify the callback function; to control the styling of a column, use the `AdvancedDataGridColumn.styleFunction` property to specify the callback function.

The callback function specified by the `AdvancedDataGrid.styleFunction` property is invoked first, followed by the function specified by the `AdvancedDataGridColumn.styleFunction` property. Therefore, the

AdvancedDataGrid control applies styles to rows first, and then to columns.

The callback function must have the following signature:

```
function_name(data:Object, column:AdvancedDataGridColumn):Object
```

The function returns an Object that contains one or more *styleName:value* pairs to specify a style setting, or null. The *styleName* field contains the name of a style property, such as `color`, and the *value* field contains the value for the style property, such as 0x00FF00. For example, you could return two styles using the following code:

```
{color:0xFF0000, fontWeight:"bold"}
```

The AdvancedDataGrid control invokes the callback function when it updates its display, such as when the control is first drawn on application start up, or when you call the `invalidateList()` method.

The examples in this section use callback functions to set the style of an AdvancedDataGrid control. All of the examples in this section use the following data from the StyleData.as file:

```
[Bindable]
private var dpADG:ArrayCollection = new ArrayCollection([
    {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
    {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99},
    {Artist:'Saner', Album:'A Child Once', Price:11.99},
    {Artist:'Saner', Album:'Helium Wings', Price:12.99},
    {Artist:'The Doors', Album:'The Doors', Price:10.99},
    {Artist:'The Doors', Album:'Morrison Hotel', Price:12.99},
    {Artist:'Grateful Dead', Album:'American Beauty', Price:11.99},
    {Artist:'Grateful Dead', Album:'In the Dark', Price:11.99},
    {Artist:'Grateful Dead', Album:'Shakedown Street', Price:11.99},
    {Artist:'The Doors', Album:'Strange Days', Price:12.99},
    {Artist:'The Doors', Album:'The Best of the Doors', Price:10.99}
]);
```

Styling rows

The following example uses the `AdvancedDataGrid.styleFunction` property to specify a callback function to apply styles to the rows of an AdvancedDataGrid control. In this example, you use Button controls to select an artist in the AdvancedDataGrid control. Then the callback function highlights in red all rows for the selected artist.

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleADGRowStyleFunc.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
      <![CDATA[
          import mx.collections.ArrayCollection;
          import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;

          // Include the data for the AdvancedDataGrid control.
          include "StyleData.as"

          // Artist name to highlight.
          protected var artistName:String;

          // Event handler to set the selected artist's name
          // based on the selected Button control.
          public function setArtistName(event:Event):void {
             artistName=Button(event.currentTarget).label;
             // Refresh row display.
             myADG.invalidateList();
          }

          // Callback function that highlights in red
          // all rows for the selected artist.
          public function myStyleFunc(data:Object,
            col:AdvancedDataGridColumn):Object {
             if (data["Artist"] == artistName)
                return {color:0xFF0000};

             // Return null if the Artist name does not match.
             return null;
          }
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        dataProvider="{dpADG}"
        styleFunction="myStyleFunc">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist"/>
            <mx:AdvancedDataGridColumn dataField="Album"/>
            <mx:AdvancedDataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:AdvancedDataGrid>

    <mx:HBox>
       <mx:Button label="Pavement" click="setArtistName(event);"/>
       <mx:Button label="Saner" click="setArtistName(event);"/>
       <mx:Button label="The Doors" click="setArtistName(event);"/>
    </mx:HBox>
</s:Application>
```

Styling columns

The following example modifies the example in the previous section to use a callback function to apply styles to the rows and one column of an AdvancedDataGrid control. In this example, the Price column displays all cells with a price less than $11.00 in green:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleADGColumnStyleFunc.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
      <![CDATA[
          import mx.collections.ArrayCollection;
          import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;

          // Include the data for the AdvancedDataGrid control.
          include "StyleData.as"

          // Artist name to highlight.
          protected var artistName:String;

          // Event handler to set the selected artist's name
          // based on the selected Button control.
          public function setArtistName(event:Event):void
          {
             artistName=Button(event.currentTarget).label;
```

```
            // Refresh row display.
            myADG.invalidateList();
         }

         // Callback function that highlights in red
         // all rows for the selected artist.
         public function myStyleFunc(data:Object,
           col:AdvancedDataGridColumn):Object
         {
            if (data["Artist"] == artistName)
               return {color:0xFF0000};

            // Return null if the Artist name does not match.
            return null;
         }
        // Callback function that hightlights in green
        // all entries in the Price column
        // with a value less than $11.00.
        public function myColStyleFunc(data:Object,
           col:AdvancedDataGridColumn):Object
        {
            if(data["Price"] <= 11.00)
                return {color:0x00FF00};

            return null;
        }
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        dataProvider="{dpADG}"
        styleFunction="myStyleFunc">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist"/>
            <mx:AdvancedDataGridColumn dataField="Album"/>
            <mx:AdvancedDataGridColumn dataField="Price"
                styleFunction="myColStyleFunc"/>
        </mx:columns>
    </mx:AdvancedDataGrid>

    <mx:HBox>
       <mx:Button label="Pavement" click="setArtistName(event);"/>
       <mx:Button label="Saner" click="setArtistName(event);"/>
       <mx:Button label="The Doors" click="setArtistName(event);"/>
    </mx:HBox>
</s:Application>
```

Using a data formatter in a column

The AdvancedDataGridColumn class contains a `formatter` property that lets you pass a class that implements the IFormatter interface to the column. This includes the Spark formatters such as CurrencyFormatter, DateTimeFormatter, and NumberFormatter, as well as subclasses of the Formatter class, including the MX CurrencyFormatter, DateFormatter, and NumberFormatter.

The formatter classes convert data into customized strings. For example, you can use the Spark or MX CurrencyFormatter class to prefix the value in the Price column with a dollar ($) sign.

You use a formatter class with a callback function specified by the `labelFunction` property or the `styleFunction` property. If you specify a callback function, Flex invokes that formatter class after invoking the callback functions.

The following example modifies the example from the section “Styling rows” on page 1402 to add a CurrencyFormatter class to the Price column to prefix the price with the $ sign:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleADGRowFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
      <![CDATA[
          import mx.collections.ArrayCollection;
          import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;

          // Include the data for the AdvancedDataGrid control.
          include "StyleData.as"

          // Artist name to highlight.
          protected var artistName:String;

          // Event handler to set the selected artist's name
          // based on the selected Button control.
          public function setArtistName(event:Event):void
          {
             artistName=Button(event.currentTarget).label;
             // Refresh row display.
             myADG.invalidateList();
          }

          // Callback function that hightlights in red
          // all rows for the selected artist.
          public function myStyleFunc(data:Object,
            col:AdvancedDataGridColumn):Object
          {
             if (data["Artist"] == artistName)
                return {color:0xFF0000};

             // Return null if the Artist name does not match.
             return null;
          }
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
```

```
        width="100%" height="100%"
        dataProvider="{dpADG}"
        styleFunction="myStyleFunc">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist"/>
            <mx:AdvancedDataGridColumn dataField="Album"/>
            <mx:AdvancedDataGridColumn width="75" dataField="Price">
                <mx:formatter>
                    <s:CurrencyFormatter useCurrencySymbol="true"/>
               </mx:formatter>
            </mx:AdvancedDataGridColumn>
        </mx:columns>
    </mx:AdvancedDataGrid>

    <s:HGroup>
       <s:Button label="Pavement" click="setArtistName(event);"/>
       <s:Button label="Saner" click="setArtistName(event);"/>
       <s:Button label="The Doors" click="setArtistName(event);"/>
    </s:HGroup>
</s:Application>
```

Selecting multiple cells and rows

All list-based controls support the `allowMultipleSelection` property. Setting the `allowMultipleSelection` property to `true` lets you select more than one item in the control at the same time. For example, the DataGrid control lets you select multiple rows so that you can drag and drop them to another DataGrid control.

The AdvancedDataGrid control adds the capability of letting you select multiple cells. You can drag the selected cells to another AdvancedDataGrid control, copy them to the clipboard, or perform some other action on the cell data.

You use the `selectionMode` property of the AdvancedDataGrid control, with the `allowMultipleSelection` property to configure multiple selection. The default value of the `selectionMode` property is `singleRow`, which means that you can select only a single row at a time. You can also set the `selectionMode` property to `singleCell`, `multipleRows` or to `multipleCells`.

Select contiguous items in the control

1 Click the first item, either a row or cell, to select it.

2 Hold down the Shift key as you select an additional item.

- If the `selectionMode` property is set to `multipleRows`, click any cell in another row to select multiple, contiguous rows.
- If the `selectionMode` property is set to `multipleCells`, click any cell to select multiple, contiguous cells.

Select discontiguous items in the control

1 Click the first item, either a row or cell, to select it.

2 Hold down the Control key as you select an additional item.

- If the `selectionMode` property is set to `multipleRows`, click any cell in another row to select that single row.
- If the `selectionMode` property is set to `multipleCells`, click any cell to select that single cell.

As you select cells or rows in the AdvancedDataGrid control, the control updates the `selectedCells` property with information about your selection. The `selectedCells` property is an Array of Objects where each Object contains a `rowIndex` and `columnIndex` property that represents the location of the selected row or cell in the control.

The value of the `selectionMode` property determines the data in the `rowIndex` and `columnIndex` properties, as the following tables describes:

Value of selectionMode property	Value of rowIndex and columnIndex properties
`none`	No selection allowed in the control, and `selectedCells` is null.
`singleRow`	Click any cell in the row to select the row. After the selection, `selectedCells` contains a single Object: `[{rowIndex:`*`selectedRowIndex`*`, columnIndex: -1}]`

`multipleRows`	Click any cell in the row to select the row.
	• While holding down the Control key, click any cell in another row to select the row for discontiguous selection.
	• While holding down the Shift key, click any cell in another row to select multiple, contiguous rows.
	After the selection, `selectedCells` contains one Object for each selected row:
	`[{rowIndex: `*`selectedRowIndex1`*`, columnIndex: -1},`
	`{rowIndex: `*`selectedRowIndex2`*`, columnIndex: -1}, ...`
	`{rowIndex: `*`selectedRowIndexN`*`, columnIndex: -1} ]`

`singleCell`	Click any cell to select the cell. After the selection, `selectedCells` contains a single Object: `[{rowIndex: `*`selectedRowIndex`*`, columnIndex:`*`selectedColIndex`*`}]`

`multipleCells`	Click any cell to select the cell.
	• While holding down the Control key, click any cell to select the cell multiple discontiguous selection.
	• While holding down the Shift key, click any cell to select multiple, contiguous cells.
	After the selection, `selectedCells` contains one Object for each selected cell:
	`[{rowIndex: `*`selectedRowIndex1`*`, columnIndex: `*`selectedColIndex1`*`},`
	`{rowIndex: `*`selectedRowIndex2`*`, columnIndex: `*`selectedColIndex2`*`},`
	`...`
	`{rowIndex: `*`selectedRowIndexN`*`, columnIndex: `*`selectedColIndexN`*`} ]`

The following example sets the `selectionMode` property to `multipleCells` to let you select multiple cells in the control. This application uses an event handler for the `keyUp` event to recognize the Control+C key combination and, if detected, copies the selected cells from the AdvancedDataGrid control to your system's clipboard.

After you copy the cells, you can paste the cells in another location in the Flex application, or paste them in another application, such as Microsoft Excel. In this example, you paste them to the TextArea control located at the bottom of the application:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/CellSelectADG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
      <![CDATA[
        import mx.collections.ArrayCollection;
        import flash.events.KeyboardEvent;
        import flash.system.System;

        include "StyleData.as"

        // Event handler to recognize when Ctrl-C is pressed,
        // and copy the selected cells to the system clipboard.
        private function myKeyUpHandler(event:KeyboardEvent):void
        {
          var keycode_c:uint = 67;
          if (event.ctrlKey && event.keyCode == keycode_c)
          {
              // Separator used between Strings sent to clipboard
              // to separate selected cells.
              var separator:String = ",";
              // The String sent to the clipboard
              var dataString:String = "";
              // Loop over the selectedCells property.
              // Data in selectedCells is ordered so that
              // the last selected cell is at the head of the list.
              // Process the data in reverse so
              // that it appears in the correct order in the clipboard.
              var n:int = event.currentTarget.selectedCells.length;
              for (var i:int = 0; i < n; i++)
              {
                  var cell:Object = event.currentTarget.selectedCells[i];
                  // Get the row for the selected cell.
                  var data:Object =
                    event.currentTarget.dataProvider[cell.rowIndex];
                  // Get the name of the field for the selected cell.
                  var dataField:String =
                    event.currentTarget.columns[cell.columnIndex].dataField;
                  // Get the cell data using the field name.
                  dataString = data[dataField] + separator + dataString;
              }
              // Remove trailing separator.
              dataString =
                dataString.substr(0, dataString.length - separator.length);
              // Write dataString to the clipboard.
              System.setClipboard(dataString);
          }
        }
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%"
        dataProvider="{dpADG}"
        selectionMode="multipleCells"
        allowMultipleSelection="true"
        keyUp="myKeyUpHandler(event);">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist"/>
            <mx:AdvancedDataGridColumn dataField="Album"/>
            <mx:AdvancedDataGridColumn dataField="Price"/>
        </mx:columns>
   </mx:AdvancedDataGrid>

   <mx:TextArea id="myTA"/>
</s:Application>
```

Hierarchical and grouped data display

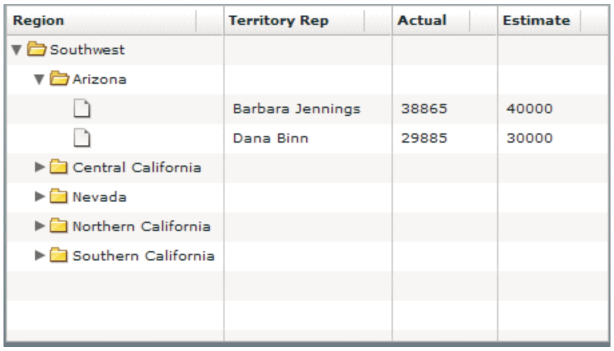
The AdvancedDataGrid control supports the display of hierarchical and grouped data. To support this display, the AdvancedDataGrid control displays a navigation tree in a column that lets you navigate the data hierarchy.

Hierarchical data is data already in a structure of parent and child data items. You can pass hierarchical data directly to the AdvancedDataGrid control. *Grouped data* is data that starts out as flat data with no inherent hierarchy. Before passing the flat data to the AdvancedDataGrid control, you specify one or more data fields that are used to group the flat data into a hierarchy.

The following code shows hierarchical data in the SimpleHierarchicalData.as file:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", children: [
     {Region:"Arizona", children: [
        {Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
        {Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]},
     {Region:"Central California", children: [
        {Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]},
     {Region:"Nevada", children: [
        {Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]},
     {Region:"Northern California", children: [
        {Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
        {Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]},
     {Region:"Southern California", children: [
        {Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
        {Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
  ]}
]);
```

Notice that the data contains a top-level object that contains a Region field and multiple second-level children. Each second-level child also contains a Region field and one or more additional children. The following example shows the AdvancedDataGrid control displaying this data:



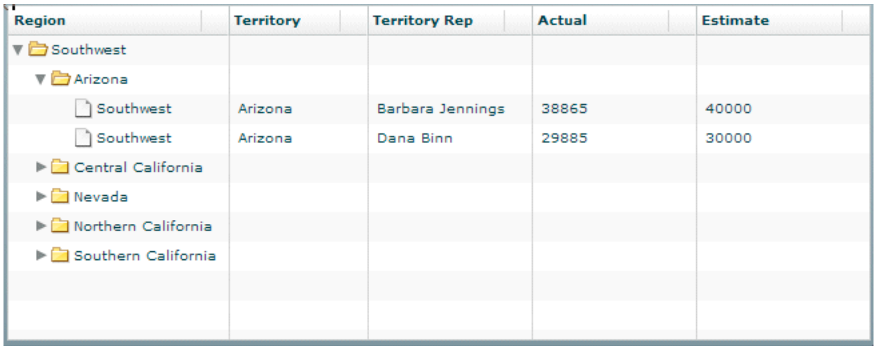
Region	Territory Rep	Actual	Estimate
▼ Southwest			
▼ Arizona			
	Barbara Jennings	38865	40000
	Dana Binn	29885	30000
▶ Central California			
▶ Nevada			
▶ Northern California			
▶ Southern California			

The code for this example is in the section “Controlling the navigation tree of the AdvancedDataGrid control” on page 1412.

To display flat data as a hierarchy, you group the rows of the flat data before passing the data to the AdvancedDataGrid control. The following code contains a variation on the hierarchical data shown in the previous image, but arranges the data in a flat data structure:

```
[Bindable]
private var dpFlat:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", Territory:"Arizona",
     Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
  {Region:"Southwest", Territory:"Arizona",
     Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000},
  {Region:"Southwest", Territory:"Central California",
     Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000},
  {Region:"Southwest", Territory:"Nevada",
     Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000},
  {Region:"Southwest", Territory:"Northern California",
     Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
  {Region:"Southwest", Territory:"Northern California",
     Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000},
  {Region:"Southwest", Territory:"Southern California",
     Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
  {Region:"Southwest", Territory:"Southern California",
     Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}
]);
```

In this example, the data contains a single level of individual records with no inherent hierarchy. To group the data, you specify one or more data fields used to arrange the data in a hierarchy. The following example shows the AdvancedDataGrid control where the flat data has been grouped by the Region field of the data:



Region	Territory	Territory Rep	Actual	Estimate
▼ Southwest				
▼ Arizona				
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
▶ Central California				
▶ Nevada				
▶ Northern California				
▶ Southern California				

The code for this example is in the section “Displaying grouped data” on page 1420.

Setting the data provider with hierarchical data

To configure the AdvancedDataGrid control to display hierarchical data and the navigation tree, you pass to the `dataProvider` property an instance of the HierarchicalData class or of the GroupingCollection2 class. Use the HierarchicalData class when your data is already arranged in a hierarchy. Use the GroupingCollection2 class when your data is in a flat structure. As part of configuring an instance of the GroupingCollection2 class, you specify one or more data fields that are used to arrange the flat data in a hierarchy.

When using the AdvancedDataGrid control with hierarchical data, you might have to free memory when you replace the data provider of the control. For more information, see the article Free memory when you replace the data provider of the AdvancedDataGrid control.

For more information on displaying hierarchical data, see “Displaying hierarchical data” on page 1418. For more information on displaying grouped data, see “Displaying grouped data” on page 1420.

You can create an instance of the HierarchicalData class or an instance of the GroupingCollection2 class from any data that you can use as a data provider. However, the AdvancedDataGrid control modifies its internal representation of the data as follows:

- An Array is represented internally by the AdvancedDataGrid control as an instance of the ArrayCollection class.
- An ArrayCollection class is represented internally by the AdvancedDataGrid control as an instance of the ArrayCollection class.
- A String that contains valid XML text is represented internally by the AdvancedDataGrid control as an instance of the XMLListCollection class.
- An XMLNode class or an XMLList class is represented internally by the AdvancedDataGrid control as an instance of the XMLListCollection class.
- Any object that implements the ICollectionView interface is represented internally by the AdvancedDataGrid control as an instance of the ICollectionView interface.
- An object of any other data type is wrapped in an Array instance with the object as its sole entry.

For example, you use an Array to create an instance of the HierarchicalData class, and then pass that HierarchicalData instance to the `AdvancedDataGrid.dataProvider` property. If you read the data back from the `AdvancedDataGrid.dataProvider` property, it is returned as an ArrayCollection instance.

Calling validateNow() after setting the data provider

In some situations, you may set the data provider of the AdvancedDataGrid control to hierarchical or grouped data, and then immediately try to perform an action based on the new data provider. This typically occurs when you set the `dataProvider` property in ActionScript, as the following example shows:

```
adg.dataProvider = groupedCollection;
adg.expandAll();
```

In this example, the call to `expandAll()` fails because the AdvancedDataGrid control is in the process of setting the `dataProvider` property, and the `expandAll()` method either processes the old value of the `dataProvider` property, if one existed, or does nothing.

In this situation, you must insert the `validateNow()` method after setting the data provider. The `validateNow()` method validates and updates the properties and layout of the control, and then redraws it, if necessary. The following example inserts the `validateNow()` method before the `expandAll()` method:

```
adg.dataProvider = groupedCollection;
adg.validateNow();
adg.expandAll();
```

Do not insert the `validateNow()` method every time you set the `dataProvider` property because it can affect the performance of your application; it is only required in some situations when you attempt to perform an operation on the control immediately after setting the `dataProvider` property.

Controlling the navigation tree of the AdvancedDataGrid control

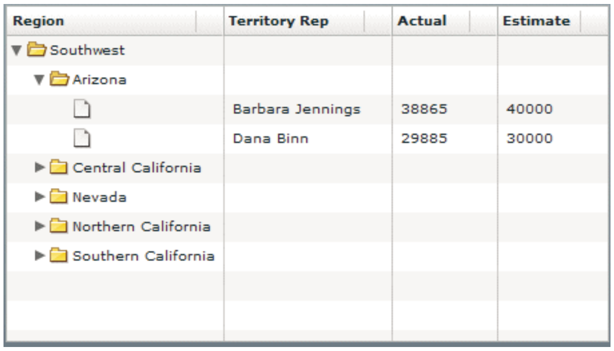
The AdvancedDataGrid control is sometimes referred to as a *tree datagrid* because a column of the control uses an expandable tree to determine which rows are currently visible in the control. Often, the tree appears in the left-most column of the control, where the first column of the control is associated with a field of the control's data provider. That data field of the data provider provides the text for the labels of the tree nodes.

In the following example, you populate the AdvancedDataGrid control with the hierarchal data structure shown in "Hierarchical and grouped data display" on page 1409. The data contains a top-level object that contains a Region field, and multiple second-level children. Each second-level child also contains a Region field and one or more children.

The AdvancedDataGrid control in this example defines four columns to display the data: Region, Territory Rep, Actual, and Estimate.

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleHierarchicalADG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

The following image shows the AdvancedDataGrid control created by this example. The control displays a folder icon to represent branch nodes of the tree, and a file icon to represent leaf nodes. The first column of the control is associated with the Region field of the data provider, so the tree labels display the value of the Region field.



Region	Territory Rep	Actual	Estimate
▼ Southwest			
▼ Arizona			
	Barbara Jennings	38865	40000
	Dana Binn	29885	30000
▶ Central California			
▶ Nevada			
▶ Northern California			
▶ Southern California			

Notice that the leaf icon for the tree does not show a label. This is because the individual records for each territory representative do not contain a Region field.

While the tree is often positioned in the left-most column of the control, you can use the `treeColumn` property to specify any column in the control, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/HierarchicalADGCategoriesTreeColumn.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "HierarchicalDataCategories.as";
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid
        width="100%" height="100%"
        treeColumn="{rep}">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"
                childrenField="categories"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn id="rep"
                dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

Setting navigation tree icons and labels

The navigation tree lets you control the icons and labels used for the branch and leaf nodes. For example, you can display a tree of labels with no icons, a tree with just folder icons, a tree with no labels at all, or a tree in its own column that is not associated with any data field.

The following table describes the style properties of the AdvancedDataGrid control that you use to set the tree icons:

Style property	Description
`defaultLeafIcon`	Specifies the leaf icon.
`disclosureClosedIcon`	Specifies the icon that is displayed next to a closed branch node. The default icon is a black triangle.
`disclosureOpenIcon`	Specifies the icon that is displayed next to an open branch node. The default icon is a black triangle.
`folderClosedIcon`	Specifies the folder closed icon for a branch node.
`folderOpenIcon`	Specifies the folder open icon for a branch node.

The following example sets the default leaf icon to null to hide it, and uses custom icons for the folder open and closed icons:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleHierarchicalADGTreeIcons.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%"
        defaultLeafIcon="{null}"
        folderOpenIcon="@Embed(source='assets/folderOpenIcon.jpg')"
        folderClosedIcon="@Embed(source='assets/folderClosedIcon.jpg')">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

You can also specify the styles by using the `setStyle()` method, and by using an `<fx:Style>` tag, as the following example shows:

```
<fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    s|AdvancedDataGrid {
        defaultLeafIcon:ClassReference(null);
        folderOpenIcon:Embed(source='assets/folderOpenIcon.jpg');
        folderClosedIcon:Embed(source='assets/folderClosedIcon.jpg');
    }
</fx:Style>
```

Using the groupIconFunction and groupLabelFunction properties

Use the `groupIconFunction` and `groupLabelFunction` properties of the AdvancedDataGrid class to define callback functions to control the display of the leaf nodes of the navigation tree. The callback function specified by the `groupIconFunction` property controls the icon, and the callback function specified by the `groupLabelFunction` controls the label.

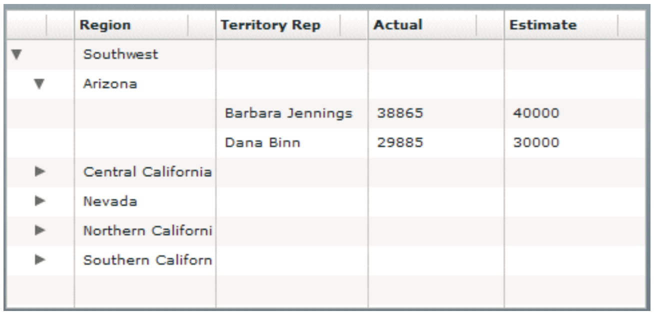
The following example uses the `groupIconFunction` property to display a custom icon for the top level node of the navigation tree, and display the default icon for all other leaf nodes of the tree:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleHierarchicalADGGroupIcon.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
         // Embed the icon for the groups.
         [Bindable]
         [Embed(source="assets/topTreeIcon.png")]
         public var icn:Class;

         // Define the groupIconFunction callback function.
         public function myIconFunc(item:Object, depth:int):Class {
             if(depth == 1)
                 // If this is the top-level of the tree, return the icon.
                 return icn;
             else
                 // If this is any other level, return null.
                 return null;
         }
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid
        width="100%" height="100%"
        groupIconFunction="myIconFunc">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

Creating a separate column for the navigation tree

In the examples in “Hierarchical and grouped data display” on page 1409, the first column also displays the Region field of the data. Therefore, as you expand the nodes of the navigation tree, the label of each tree node corresponds to the value of the Region field in the data provider for that row. For the leaf nodes of the tree, the data provider does not contain a value for the Region field, so the labels for the leaf nodes are blank.

You can also put the navigation tree in its own column, where the column is not associated with a data field, as the following example shows:



	Region	Territory Rep	Actual	Estimate
▼	Southwest			
▼	Arizona			
		Barbara Jennings	38865	40000
		Dana Binn	29885	30000
▶	Central California			
▶	Nevada			
▶	Northern Californi			
▶	Southern Californ			

This example does not associate a data field with the column that contains the tree, so the tree icons appear with no label. This example also sets the `folderClosedIcon`, `folderOpenIcon`, and `defaultLeafIcon` properties of the AdvancedDataGrid control to null, but does display the disclosure icons so that the user can still open and close tree nodes.

The following code implements this example. Notice that the first column is not associated with a data field. Because it is the first column in the data grid, the AdvancedDataGrid control automatically uses it to display the navigation tree.

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleHierarchicalADGTreeColumn.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%"
        folderClosedIcon="{null}"
        folderOpenIcon="{null}"
        defaultLeafIcon="{null}">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn headerText="" width="50"/>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

Displaying hierarchical data

Hierarchical data is data in a structured format where the data is already arranged in a hierarchy. To display hierarchical data in the AdvancedDataGrid control, you set the data provider of the control to an instance of the HierarchicalData class. The structure of the data in the data provider defines how the AdvancedDataGrid control displays the data.

Defining hierarchical data with an ArrayCollection

Using an ArrayCollection is a common way to create hierarchical data, as the following example shows in the SimpleHierarchicalData.as file. In this example, the data has three levels – a root level and two child levels:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", children: [
     {Region:"Arizona", children: [
        {Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
        {Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]},
     {Region:"Central California", children: [
        {Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]},
     {Region:"Nevada", children: [
        {Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]},
     {Region:"Northern California", children: [
        {Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
        {Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]},
     {Region:"Southern California", children: [
        {Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
        {Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
  ]}
]);
```

This example uses the `children` keyword in the ArrayCollection definition to define the data hierarchy. The `children` keyword is the default keyword used by the HierarchicalData class to define the hierarchy.

You can use a different keyword to define the hierarchy. The following example shows the HierarchicalDataCategories.as file, which uses the `categories` keyword:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", categories: [
     {Region:"Arizona", categories: [
        {Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
        {Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]},
     {Region:"Central California", categories: [
        {Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]},
     {Region:"Nevada", categories: [
        {Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]},
     {Region:"Northern California", categories: [
        {Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
        {Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]},
     {Region:"Southern California", categories: [
        {Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
        {Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
  ]}
]);
```

Use the `HierarchicalData.childrenField` property to specify the name of the field that defines the hierarchy, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/HierarchicalADGCategories.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "HierarchicalDataCategories.as";
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"
                childrenField="categories"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

Displaying hierarchical XML data

The examples in the previous section use an ArrayCollection to populate the AdvancedDataGrid control. You can also populate the control with hierarchical XML data. The following example modifies the data from the previous section to format it as XML, and then passes that data to the AdvancedDataGrid control:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleHierarchicalADGXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.HierarchicalData;
            import mx.collections.XMLListCollection;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:XMLList id="dpHierarchyXML" >
            <Region Region="Southwest">
                <Region Region="Arizona">
                    <Territory_Rep Territory_Rep="Barbara Jennings"
                        Actual="38865" Estimate="40000"/>
                    <Territory_Rep Territory_Rep="Dana Binn"
                        Actual="29885" Estimate="30000"/>
                </Region>
                <Region Region="Central California">
                    <Territory_Rep Territory_Rep="Joe Smith"
                        Actual="29134" Estimate="30000"/>
                </Region>
                <Region Region="Nevada">
                    <Territory_Rep Territory_Rep="Bethany Pittman"
                        Actual="52888" Estimate="45000"/>
                </Region>
                <Region Region="Northern California">
                    <Territory_Rep Territory_Rep="Lauren Ipsum"
                        Actual="38805" Estimate="40000"/>
                    <Territory_Rep Territory_Rep="T.R. Smith"
                        Actual="55498" Estimate="40000"/>
                </Region>
                <Region Region="Southern California">
                    <Territory_Rep Territory_Rep="Alice Treu"
                        Actual="44985" Estimate="45000"/>
                    <Territory_Rep Territory_Rep="Jane Grove"
                        Actual="44913" Estimate="45000"/>
                </Region>
            </Region>
        </fx:XMLList>
    </fx:Declarations>
    <mx:AdvancedDataGrid width="100%" height="100%"
        dataProvider="{new HierarchicalData(dpHierarchyXML)}">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="@Region"
                headerText="Region"/>
            <mx:AdvancedDataGridColumn dataField="@Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="@Actual"
                headerText="Actual"/>
            <mx:AdvancedDataGridColumn dataField="@Estimate"
                headerText="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

Displaying grouped data

Grouped data is flat data that you arrange in a hierarchy for display in the AdvancedDataGrid control. To group your data, you specify one or more data fields that collect the data into the hierarchy.

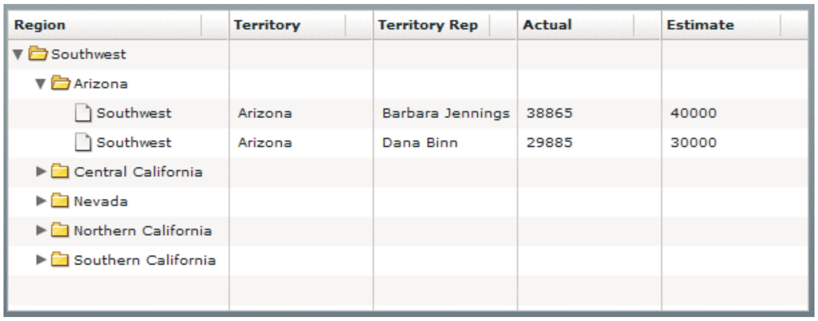
To populate the AdvancedDataGrid control with grouped data, create an instance of the GroupingCollection2 class from your flat data, and then pass that GroupingCollection2 instance to the data provider of the AdvancedDataGrid control. When you create the instance of the GroupingCollection2 from your flat data, you specify the fields for the data that are used to create the hierarchy.

Note: *In the previous release of Flex, you used the GroupingCollection class with the AdvancedDataGrid control. The GroupingCollection2 class is new for Flex 4 and provides better performance than GroupingCollection.*

Most of the examples in this section use the following flat data to create an instance of the GroupingCollection2 class:

```
[Bindable]
private var dpFlat:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000},
  {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000},
  {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}
]);
```

The following image shows an AdvancedDataGrid control that uses this data as input. This example specifies two fields that are used to group the data: Region and Territory. The first column of the AdvancedDataGrid control is associated with the Region field, so the navigation tree uses the Region field to define the labels for the leaf nodes of the tree.



Region	Territory	Territory Rep	Actual	Estimate
▼ Southwest				
▼ Arizona				
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
▶ Central California				
▶ Nevada				
▶ Northern California				
▶ Southern California				

The following code implements this example:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleGroupADGMXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection2 id="gc" source="{dpFlat}">
                <mx:grouping>
                    <mx:Grouping label="Region">
                        <mx:GroupingField name="Region"/>
                        <mx:GroupingField name="Territory"/>
                    </mx:Grouping>
                </mx:grouping>
            </mx:GroupingCollection2>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

The GroupingCollection2 instance reformats the data based on these fields so that internally it is represented by the GroupingCollection2 instance, as the following example shows:

```
[{GroupLabel:"Southwest", children:[
    {GroupLabel:"Arizona", children:[
      {Region:"Southwest", Territory:"Arizona",
        Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
      {Region:"Southwest", Territory:"Arizona",
        Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]}
    {GroupLabel:"Central California", children:[
      {Region:"Southwest", Territory:"Central California",
        Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]}
    {GroupLabel:"Nevada", children:[
      {Region:"Southwest", Territory:"Nevada",
        Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]}
    {GroupLabel:"Northern California", children:[
      {Region:"Southwest", Territory:"Northern California",
        Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
      {Region:"Southwest", Territory:"Northern California",
        Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]}
    {GroupLabel:"Southern California", children:[
      {Region:"Southwest", Territory:"Southern California",
        Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
      {Region:"Southwest", Territory:"Southern California",
        Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
]}]
```

Notice that the representation consists of a data hierarchy based on the Region and Territory fields of the flat data. The GroupingCollection2 instance still contains the original rows of the input flat data, but those rows are now arranged in a hierarchy based on the grouping fields.

Using synchronous or asynchronous refresh with the AdvancedDataGrid control

The `GroupingCollection2.refresh()` method applies the settings of the GroupingCollection2 class to the data. You must call this method any time you modify the GroupingCollection2 class, such as by setting the `grouping`, `source`, or `summaries` properties of the GroupingCollection2 class. You also call the `GroupingCollection2.refresh()` method when you modify a GroupingField of the GroupingCollection2 class, such as by changing the `caseInsensitive`, `compareFunction`, or `groupingFunction` properties.

Notice that in the previous example, the AdvancedDataGrid control calls the `GroupingCollection2.refresh()` method in response to the `initialize` event of the AdvancedDataGrid control.

The `GroupingCollection2.refresh()` method has the following signature:

```
public function refresh(async:Boolean = false, dispatchCollectionEvents:Boolean =
false):Boolean
```

Set the `async` parameter to `false`, the default value, for synchronous refresh of the groups and any group summaries. Set it to `true` for asynchronous refresh.

For synchronous refresh, all groups and summaries are updated together before the method returns. That means your application cannot perform other processing operations for the duration of the call. It also prevents the user from interacting with the control until the refresh completes. Synchronous refresh is optimized for small data sets. Because a synchronous refresh updates all groups and summaries together, it has better performance than an asynchronous refresh. However, for large data sets, users might notice that the application pauses for the duration of the refresh.

In asynchronous refresh, all groups and summaries are updated individually. The `refresh()` method returns before the groups and summaries are updated so that your application can continue execution. Also, the control is updated during the refresh so that the user can continue to interact with it. For example, in asynchronous refresh, if the `displayItemsExpanded` property is `true`, the navigation tree is updated as new groups are created.

The overhead of updating groups and summaries individually, rather than all at once, makes an asynchronous refresh take longer than a synchronous one. However, for large data sets, your application continues to operate during the refresh.

The `dispatchCollectionEvents` parameter configures the dispatching of events in synchronous mode. If `false`, the default value, then no events are dispatched during the refresh. This setting provides the best performance for creating groups and summaries. If `true`, then events are dispatched as groups and summaries are calculated to update the control. However, the navigation tree is not updated as new groups are created.

For an asynchronous refresh, the `dispatchCollectionEvents` parameter is always set to `true`.

Using the default MXML property of the GroupingCollection2 class

The `grouping` property is the default MXML property of the GroupingCollection2 class, so you can rewrite the previous example as follows:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleGroupADGMXMLDefaultProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection2 id="gc" source="{dpFlat}">
                    <mx:Grouping label="Region">
                        <mx:GroupingField name="Region"/>
                        <mx:GroupingField name="Territory"/>
                    </mx:Grouping>
            </mx:GroupingCollection2>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

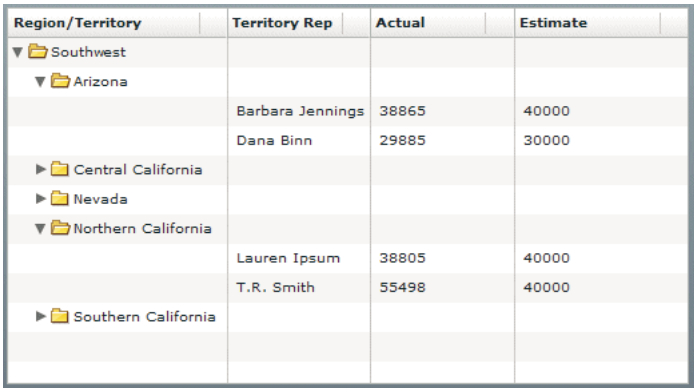
Setting the name of the GroupLabel field

By default, GroupLabel is the field name for the data items added to the flat data to create the hierarchy. For an example of the data structure created by the GroupingCollection2 class that uses the default field name, see “Displaying grouped data” on page 1420.

Use the `Grouping.label` property to specify a different name so that you can create a different group label for each level of the generated hierarchy. You should always set the `Grouping.label` property for the top level in your hierarchy. That property is required to sort the nodes of the navigation tree.

Creating a column for the GroupLabel field

The AdvancedDataGrid control uses the GroupLabel field to define the labels for the branch nodes of the navigation tree. One option when displaying grouped data is to create a column for the top-level data items that the grouping fields create. For example, you group your flat data by the Region and Territory fields. These fields appear as the labels for the branch nodes of the navigation tree, so you omit separate columns for those fields, as the following example shows:



Region/Territory	Territory Rep	Actual	Estimate
▼ Southwest			
▼ Arizona			
	Barbara Jennings	38865	40000
	Dana Binn	29885	30000
▶ Central California			
▶ Nevada			
▼ Northern California			
	Lauren Ipsum	38805	40000
	T.R. Smith	55498	40000
▶ Southern California			

The following code creates this example. Notice that this example includes an AdvancedDataGridColumn instance for the GroupLabel field, and no column definitions for the Region and Territory fields.

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleGroupADGGroupLabelMXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        defaultLeafIcon="{null}"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection2 id="gc" source="{dpFlat}">
                <mx:Grouping>
                    <mx:GroupingField name="Region"/>
                    <mx:GroupingField name="Territory"/>
                </mx:Grouping>
            </mx:GroupingCollection2>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="GroupLabel"
                headerText="Region/Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

Creating groups in ActionScript

The example in the previous section created the groups in MXML. However, you could let the user define the grouping at run time. The following example creates the groups in ActionScript by using an event handler:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleGroupADG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
      <![CDATA[
        import mx.collections.GroupingField;
        import mx.collections.Grouping;
        import mx.collections.GroupingCollection2;
        import mx.collections.ArrayCollection;

        include "SimpleFlatData.as";
        [Bindable]
        public var myGColl:GroupingCollection2 = new GroupingCollection2();

        private var myGrp:Grouping = new Grouping();
        private function initDG():void {
            // Initialize the GroupingCollection2 instance.
            myGColl.source = dpFlat;

            // The Grouping instance defines the grouping fields
            // in the collection, and the order of the groups
            // in the hierarchy.
            myGrp.fields =
              [new GroupingField("Region"), new GroupingField("Territory")];
            myGrp.label = "Region";

            // The grouping property contains a Grouping instance.
            myGColl.grouping = myGrp;
            // Specify the GroupedCollection as the data provider for
            // the AdvancedDataGrid control.
            myADG.dataProvider=myGColl;

            // Refresh the display.
            myGColl.refresh();
        }
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        creationComplete="initDG();">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

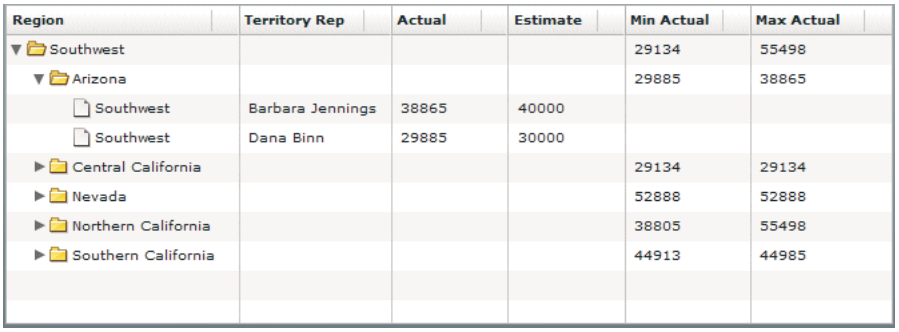
Creating summary rows

Summary data provides a way for you to extract information from your data for display in the AdvancedDataGrid control. For example, you can add summary data that contains the average value of a field across all rows of your data, that contains the maximum field value, the minimum field value, or that contains other types of summaries.

***Note:** Summary data is not supported for hierarchical data represented by the HierarchicalData class. You can only create summary data for data represented by the GroupingCollection2 class.*

You create summary data about your groups by using the `summaries` property of the GroupingField class. You can display the summary data in an existing row of the AdvancedDataGrid control, or display it in a separate row.

The following example shows an AdvancedDataGrid control that displays two summary fields, Min Actual and Max Actual:



Region	Territory Rep	Actual	Estimate	Min Actual	Max Actual
▼ Southwest				29134	55498
▼ Arizona				29885	38865
Southwest	Barbara Jennings	38865	40000		
Southwest	Dana Binn	29885	30000		
▶ Central California				29134	29134
▶ Nevada				52888	52888
▶ Northern California				38805	55498
▶ Southern California				44913	44985

The Min Actual and Max Actual fields in the top row correspond to summaries for all rows in the group, and the Min Actual and Max Actual fields for each Territory correspond to summaries for all rows in the territory subgroup.

The following code creates this control:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SummaryGroupADG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection2 id="gc" source="{dpFlat}">
                <mx:Grouping label="Region">
                    <mx:GroupingField name="Region">
                        <mx:summaries>
                          <mx:SummaryRow summaryPlacement="group">
                            <mx:fields>
                                <mx:SummaryField2 dataField="Actual"
                                    label="Min Actual" summaryOperation="MIN"/>
                                <mx:SummaryField2 dataField="Actual"
```

```
                                    label="Max Actual" summaryOperation="MAX"/>
                            </mx:fields>
                          </mx:SummaryRow>
                        </mx:summaries>
                    </mx:GroupingField>
                    <mx:GroupingField name="Territory">
                        <mx:summaries>
                          <mx:SummaryRow summaryPlacement="group">
                            <mx:fields>
                                <mx:SummaryField2 dataField="Actual"
                                    label="Min Actual" summaryOperation="MIN"/>
                                <mx:SummaryField2 dataField="Actual"
                                    label="Max Actual" summaryOperation="MAX"/>
                            </mx:fields>
                          </mx:SummaryRow>
                        </mx:summaries>
                    </mx:GroupingField>
                </mx:Grouping>
            </mx:GroupingCollection2>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn dataField="Min Actual"/>
            <mx:AdvancedDataGridColumn dataField="Max Actual"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

Notice in this example that you use the `GroupingField.summaries` property to specify the SummaryRow instance. The SummaryRow instance contains all the information about a data summary. In this example, you use the `summaryPlacement` property to add the summary data to the grouped data. Optionally, you can add new rows to contain the summary data.

Each SummaryRow instance specifies one or more SummaryField2 instances that create the data summary. In this example, you use the `dataField` property to specify that the summaries are determined by the data in the Actual data field, the `label` property to specify the name of the data field created to hold the summary data, and the `summaryOperation` property to specify how to create the summary for numeric fields. You can specify one of the following values: `SUM`, `MIN`, `MAX`, `AVG`, or `COUNT`.

Note: *In the previous release of Flex, you used the SummaryField class to create summary data. The SummaryField2 class is new for Flex 4 and provides better performance than SummaryField.*

Internally, the GroupingCollection2 class represents the grouped data, with the new summary data fields, as the following example shows:

```
[{GroupLabel:"Southwest", Max Actual:55498, Min Actual:29134, children:[
  {GroupLabel:"Arizona", Max Actual:38865, Min Actual:29885, children:[
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]}
  {GroupLabel:"Central California", Max Actual:29134, Min Actual:29134, children:[
    {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]}
  {GroupLabel:"Nevada", Max Actual:52888, Min Actual:52888, children:[
    {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]}
  {GroupLabel:"Northern California", Max Actual:55498, Min Actual:38805, children:[
    {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
    {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]}
  {GroupLabel:"Southern California", Max Actual:44985, Min Actual:44913, children:[
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
]}]
```

Using the default properties of the GroupingField and SummaryRow classes

The `GroupingField.summaries` property is the default property for the GroupingField class, and the `SummaryRow.fields` property is the default property for the SummaryRow class; therefore, you can omit those properties in your code and rewrite the previous example, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SummaryGroupADGDefProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection2 id="gc" source="{dpFlat}">
                <mx:Grouping label="Region">
                    <mx:GroupingField name="Region">
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField2 summaryOperation="MIN"
                                    dataField="Actual" label="Min Actual"/>
                                <mx:SummaryField2 summaryOperation="MAX"
                                    dataField="Actual" label="Max Actual"/>
                          </mx:SummaryRow>
                    </mx:GroupingField>
                    <mx:GroupingField name="Territory">
```

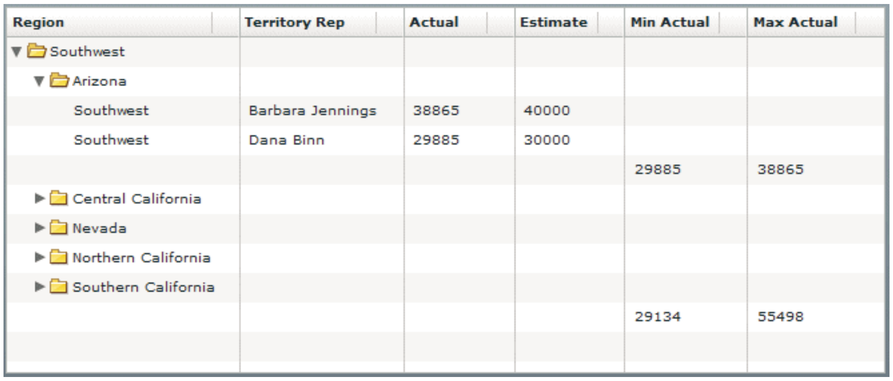
```
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField2 summaryOperation="MIN"
                                    dataField="Actual" label="Min Actual"/>
                                <mx:SummaryField2 summaryOperation="MAX"
                                    dataField="Actual" label="Max Actual"/>
                          </mx:SummaryRow>
                    </mx:GroupingField>
                </mx:Grouping>
            </mx:GroupingCollection2>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn dataField="Min Actual"/>
            <mx:AdvancedDataGridColumn dataField="Max Actual"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</s:Application>
```

Specifying the display location of the summary data in the AdvancedDataGrid control

The SummaryRow class contains the `summaryPlacement` property that determines where the summary data appears in the AdvancedDataGrid control. The following table describes the values of the `summaryPlacement` property:

Value of summaryPlacement property	Description
`first`	Creates a summary row as the first row in the group.
`last`	Creates a summary row as the last row in the group.
`group`	Adds the summary data to the row corresponding to the group.

The previous example named SummaryGroupADG.mxml shows an example of summary data added to the group by specifying `group` as the value of the `summaryPlacement` property. The following example shows the same grouped data but with the `summaryPlacement` property set to `last`:



Region	Territory Rep	Actual	Estimate	Min Actual	Max Actual
▼ Southwest					
▼ Arizona					
Southwest	Barbara Jennings	38865	40000		
Southwest	Dana Binn	29885	30000		
				29885	38865
▶ Central California					
▶ Nevada					
▶ Northern California					
▶ Southern California					
				29134	55498

You can specify multiple values to the `summaryPlacement` property, separated by a space; for example, a value of `last group` specifies to show the summary at the group level and as the last row of the group.

Internally, the GroupingCollection2 represents the grouped data, with the new summary data fields, as the following example shows:

```
[{GroupLabel:"Southwest", children:[
  {GroupLabel:"Arizona", children:[
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000},
    {Max Actual:38865, Min Actual:29885}]}
  {GroupLabel:"Central California", children:[
    {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000},
    {Max Actual:29134, Min Actual:29134}]}
  {GroupLabel:"Nevada", children:[
    {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000},
    {Max Actual:52888, Min Actual:52888}]}
  {GroupLabel:"Northern California", children:[
    {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
    {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000},
    {Max Actual:55498, Min Actual:38805}]}
  {GroupLabel:"Southern California", children:[
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000},
    {Max Actual:44985, Min Actual:44913}]}
  {Max Actual:55498, Min Actual:29134}
]}]
```

Notice that a new row is added to the entire group to hold the summary data, and a new row is added to each subgroup to hold its summary data.

Creating multiple summaries

You can specify multiple SummaryRow instances in a single GroupingField instance. In the following example, you create summary data to define the following fields: Min Actual, Max Actual, Min Estimate, and Max Estimate for the Region group.

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SummaryGroupADGMultipleSummaries.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection2 id="gc" source="{dpFlat}">
                <mx:Grouping label="Region">
                    <mx:GroupingField name="Region">
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField2 summaryOperation="MIN"
                                   dataField="Actual" label="Min Actual"/>
                                <mx:SummaryField2 summaryOperation="MAX"
                                   dataField="Actual" label="Max Actual"/>
                          </mx:SummaryRow>
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField2 summaryOperation="MIN"
                                   dataField="Estimate" label="Min Estimate"/>
                                <mx:SummaryField2 summaryOperation="MAX"
                                   dataField="Estimate" label="Max Estimate"/>
                          </mx:SummaryRow>
                    </mx:GroupingField>
                    <mx:GroupingField name="Territory"/>
                </mx:Grouping>
            </mx:GroupingCollection2>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn dataField="Min Actual"/>
            <mx:AdvancedDataGridColumn dataField="Max Actual"/>
            <mx:AdvancedDataGridColumn dataField="Min Estimate"/>
            <mx:AdvancedDataGridColumn dataField="Max Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

Creating a custom summary function

Use the `SummaryRow.summaryObjectFunction` property and the `SummaryField2.summaryOperation` property to add custom summary logic to your application. Your custom summary can perform many types of actions, but the result of the summary calculation must be of type Number.

The `SummaryRow.summaryObjectFunction` property defines a callback function that defines an instance of the SummaryObject class that collects the summary data. The AdvancedDataGrid control adds the SummaryObject instance to the data provider to display the summary data in the control. Therefore, you define within the SummaryObject instance the properties to display. In the following example, the callback function adds a property named Territory_Rep. This field appears in the Territory Rep column of the control for the summary row.

The `SummaryField2.summaryOperation` property takes an object that implements the mx.collections.ISummaryCalculator interface. That interface defines the following groups of methods:

- Implement the `summaryCalculationBegin()`, `calculateSummary()`, and `returnSummary()` methods to compute a summary of the values.
- Implement the `summaryOfSummaryCalculationBegin()`, `calculateSummaryOfSummary()`, and `returnSummaryOfSummary()` methods to compute a summary of summary values.

The following definition for the class MySummaryCalculator implements the ISummaryCalculator interface to create a class to summarize a field in every other row of a group:

```
package myComponents
{
    // dpcontrols/adg/myComponents/MySummaryCalculator.as
    import mx.collections.ISummaryCalculator;
    import mx.collections.SummaryField2;

    public class MySummaryCalculator implements ISummaryCalculator
    {
        public function MySummaryCalculator() {
        }

        // Define the summary object.
        protected var summObj:Object = new Object();
        // Define a var to hold the record count.
        protected var count:int;

        public function summaryCalculationBegin(field:SummaryField2):Object {
            // Initialize the variables.
            summObj.oddCount = 0;
            count = 1;
            return summObj;
        }

      public function calculateSummary(data:Object, field:SummaryField2, rowData:Object):void {
            // Only add the current value for every other row.
            if (count % 2 != 0)
            {
                summObj.oddCount = summObj.oddCount + rowData[field.dataField];
            }
            count++;
        }
        public function returnSummary(data:Object, field:SummaryField2):Number {
            // Return the summary value.
            return summObj.oddCount;
        }

        // Implement these methods if you are creating a summary of summaries.
        public function summaryOfSummaryCalculationBegin(value:Object,
field:SummaryField2):Object {
            return null;
        }

        public function calculateSummaryOfSummary(value:Object, newValue:Object,
field:SummaryField2):void {
        }

        public function returnSummaryOfSummary(value:Object, field:SummaryField2):Number {
            return 0;
        }
    }
}
```

Notice that this class implements the `summaryOfSummaryCalculationBegin()`, `calculateSummaryOfSummary()`, and `returnSummaryOfSummary()` methods but leaves the implementation empty.

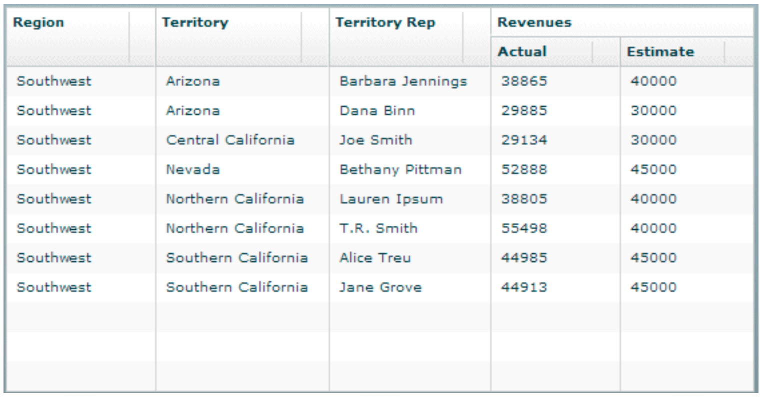
The following example implements callback functions for the `summaryObjectFunction` and use the MySummaryCalculator class to calculate a summary of the actual sales revenue for every other row of each territory:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SummaryGroupADGCustomSummary.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.IViewCursor;
            import mx.collections.SummaryObject;
            import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;
            import mx.collections.ArrayCollection;

            import myComponents.MySummaryCalculator;

            include "SimpleFlatData.as"

            // Callback function to create
            // the SummaryObject used to hold the summary data.
            private function summObjFunc():SummaryObject {

                // Define the object containing the summary data.
                var obj:SummaryObject = new SummaryObject();
                // Add a field containing a value for the Territory_Rep column.
                obj.Territory_Rep = "Alternating Reps";

                return obj;
            }
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
```

```
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection2 id="gc" source="{dpFlat}">
                <mx:Grouping label="Region">
                   <mx:GroupingField name="Region"/>
                   <mx:GroupingField name="Territory">
                      <mx:summaries>
                         <mx:SummaryRow summaryObjectFunction="summObjFunc"
                            summaryPlacement="first">
                            <mx:fields>
                               <mx:SummaryField2 dataField="Actual"
                                summaryOperation="{new myComponents.MySummaryCalculator()}"/>
                            </mx:fields>
                         </mx:SummaryRow>
                      </mx:summaries>
                   </mx:GroupingField>
                </mx:Grouping>
            </mx:GroupingCollection2>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</s:Application>
```

Creating column groups

You collect multiple columns under a single column heading by using column groups, as the following example shows:



Region	Territory	Territory Rep	Revenues	
			Actual	Estimate
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
Southwest	Central California	Joe Smith	29134	30000
Southwest	Nevada	Bethany Pittman	52888	45000
Southwest	Northern California	Lauren Ipsum	38805	40000
Southwest	Northern California	T.R. Smith	55498	40000
Southwest	Southern California	Alice Treu	44985	45000
Southwest	Southern California	Jane Grove	44913	45000

In this example, you use flat data to populate the data grid, and group the Actual and Estimate columns under a single column named Revenues.

To group columns in an AdvancedDataGrid control, you must do the following:

- Use the `AdvancedDataGrid.groupedColumns` property, rather than the `AdvancedDataGrid.columns` property, to specify the columns.

- Use the AdvancedDataGridColumnGroup class to specify the column groups.

The following code implements the AdvancedDataGrid control shown in the previous figure:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupADG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Import the data used by the AdvancedDataGrid control.
            include "SimpleFlatData.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</s:Application>
```

The `groupedColumns` property contains instances of the AdvancedDataGridColumn class and of the AdvancedDataGridColumn class. Instances of the AdvancedDataGridColumn class appear in the control as stand-alone columns. All the columns specified in an AdvancedDataGridColumnGroup instance appear together as a grouped column.

You can add multiple groups to the control. The following example adds groups named Area and Revenues:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupADG2Groups.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Area">
                <mx:AdvancedDataGridColumn dataField="Region"/>
                <mx:AdvancedDataGridColumn dataField="Territory"/>
            </mx:AdvancedDataGridColumnGroup>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</s:Application>
```

You can nest groups so that one column group contains multiple column groups, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupADG2NestedGroups.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="All Groups">
                <mx:AdvancedDataGridColumnGroup headerText="Area">
                    <mx:AdvancedDataGridColumn dataField="Region"/>
                    <mx:AdvancedDataGridColumn dataField="Territory"/>
                </mx:AdvancedDataGridColumnGroup>
                <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                    <mx:AdvancedDataGridColumn dataField="Actual"/>
                    <mx:AdvancedDataGridColumn dataField="Estimate"/>
                </mx:AdvancedDataGridColumnGroup>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</s:Application>
```

Dragging and dropping columns in the group

By default, you can drag the columns in a group within the group to reposition them. You can also drag the entire group to reposition it in the AdvancedDataGrid control.

To disable dragging of all columns in a group, set the `AdvancedDataGridColumnGroup.childrenDragEnabled` property to `false`.

Using grouped columns with hierarchical data

You can use column groups with hierarchical data, as well as flat data. The following example modifies the example in the section "Creating a separate column for the navigation tree" on page 1416 to group the Actual and Estimates columns under the Revenues group column:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SimpleHierarchicalADGGroupCol.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
    </mx:AdvancedDataGrid>
</s:Application>
```

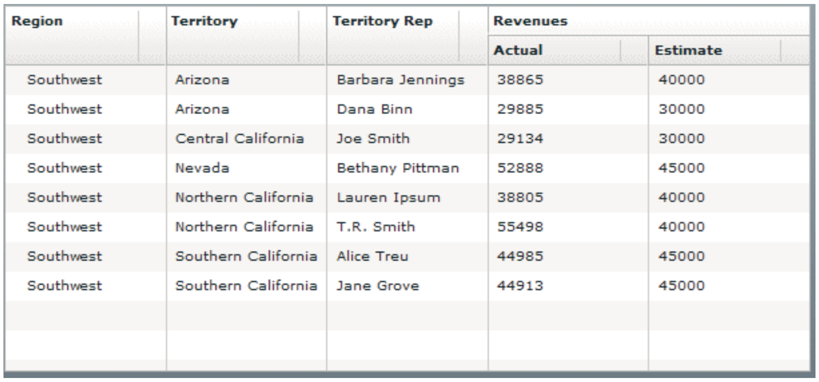
Specify a data field for the AdvancedDataGridColumnGroup class

The previous examples for column groups do not specify a data field for the AdvancedDataGridColumnGroup class. However, the AdvancedDataGridColumnGroup class is designed to work with hierarchical data. Therefore, if you specify a data field for the AdvancedDataGridColumnGroup class, it automatically creates a column group for the subfields of that data field.

In the following example, the HierarchicalDataForGroupedColumns.as file defines a hierarchical data set where the Revenues field contains two subfields, Actual and Estimate:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings",
      Revenues:{Actual:38865, Estimate:40000}},
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn",
      Revenues:{Actual:29885, Estimate:30000}},
  {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith",
      Revenues:{Actual:29134, Estimate:30000}},
  {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman",
      Revenues:{Actual:52888, Estimate:45000}},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum",
      Revenues:{Actual:38805, Estimate:40000}},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith",
      Revenues:{Actual:55498, Estimate:40000}},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu",
      Revenues:{Actual:44985, Estimate:45000}},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove",
      Revenues:{Actual:44913, Estimate:45000}}
]);
```

The following example uses this data and specifies the Revenues field as the value of the `AdvancedDataGridColumnGroup.dataField` property, and creates the following output:



Region	Territory	Territory Rep	Revenues	
			Actual	Estimate
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
Southwest	Central California	Joe Smith	29134	30000
Southwest	Nevada	Bethany Pittman	52888	45000
Southwest	Northern California	Lauren Ipsum	38805	40000
Southwest	Northern California	T.R. Smith	55498	40000
Southwest	Southern California	Alice Treu	44985	45000
Southwest	Southern California	Jane Grove	44913	45000

The following code implements this example:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupHierarchData.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "HierarchicalDataForGroupedColumns.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        defaultLeafIcon="{null}">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup dataField="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</s:Application>
```

Using item renderers with the AdvancedDataGrid control

You customize the appearance and behavior of cells in an AdvancedDataGrid control by creating custom item renderers and item editors. Use the same process for assigning item editors and item renderers to columns of the AdvancedDataGrid control as you do with the DataGrid control.

For an introduction to item renderers and item editors, see “MX item renderers and item editors” on page 1006.

The AdvancedDataGrid control adds capabilities to support item renderers. These capabilities let you perform the following actions:

- Create rows or columns not associated with data in the data provider. For example, you can create summary rows from the data provider.
- Span multiple columns with a renderer.
- Use multiple renderers in the same column. For example, when displaying hierarchical data, you can use a different renderer in the column based on the level of the row in the hierarchy.

***Note:** These capabilities support item renderers only; use item editors as you do with the DataGrid control.*

Using an item renderer

To use the item renderer capabilities of the AdvancedDataGrid control, you assign the item renderer to the AdvancedDataGrid control itself, not to a specific column, by using the `AdvancedDataGrid.rendererProviders` property. The following example assigns an item renderer to the Estimate column:

```
<mx:AdvancedDataGrid>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="Region"/>
        <mx:AdvancedDataGridColumn dataField="Territory_Rep" headerText="Territory Rep"/>
        <mx:AdvancedDataGridColumn dataField="Actual"/>
        <mx:AdvancedDataGridColumn dataField="Estimate"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            columnIndex="3"
            columnSpan="1"
            renderer="myComponents.EstimateRenderer"/>
    </mx:rendererProviders>
</mx:AdvancedDataGrid>
```

The `rendererProviders` property contains an Array of AdvancedDataGridRendererProvider instances. Each AdvancedDataGridRendererProvider instance defines the characteristics for a single item renderer.

In the previous example, the AdvancedDataGridRendererProvider instance specifies to use the EstimateRenderer for column 3, where the first column in the control is column 0, and the renderer spans a single column. If you set the `columnSpan` property to 0, the renderer spans all columns in the row.

Rather than using the column index to assign the renderer, you specify an `id` property for the column, and then use the `column` property to associate the renderer with the column, as the following example shows:

```
<mx:AdvancedDataGrid>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="Region"/>
        <mx:AdvancedDataGridColumn dataField="Territory_Rep" headerText="Territory Rep"/>
        <mx:AdvancedDataGridColumn dataField="Actual"/>
        <mx:AdvancedDataGridColumn id="estCol" dataField="Estimate"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            column="{estCol}"
            columnSpan="1"
            renderer="myComponents.EstimateRenderer"/>
    </mx:rendererProviders>
</mx:AdvancedDataGrid>
```

The `depth` property lets you associate the renderer with a specific level in the hierarchy of the navigation tree of an AdvancedDataGrid control, where the depth of the top-most node in the navigation tree is 1. The following example associates the renderer with the third level of a navigation tree:

```
    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            columnIndex="3"
            depth="3"
            columnSpan="1"
            renderer="myComponents.EstimateRenderer"/>
    </mx:rendererProviders>
```

In the previous example, the AdvancedDataGrid control uses the default renderer for the column until you expand it to the third level of the navigation tree, and then it uses EstimateRenderer component. You use the `depth` property to assign different renderers to the same column, where the `depth` property specifies the renderer to use for each level of the tree.

A renderer can span multiple columns. In the following example, you create a renderer that spans two columns:

```
<mx:AdvancedDataGrid>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="Region"/>
        <mx:AdvancedDataGridColumn dataField="Territory_Rep" headerText="Territory Rep"/>
        <mx:AdvancedDataGridColumn id="actCol" dataField="Actual"/>
        <mx:AdvancedDataGridColumn dataField="Estimate"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            column="{actCol}"
            depth="3"
            columnSpan="2"
            renderer="myComponents.SummaryRenderer"/>
    </mx:rendererProviders>
</mx:AdvancedDataGrid>
```

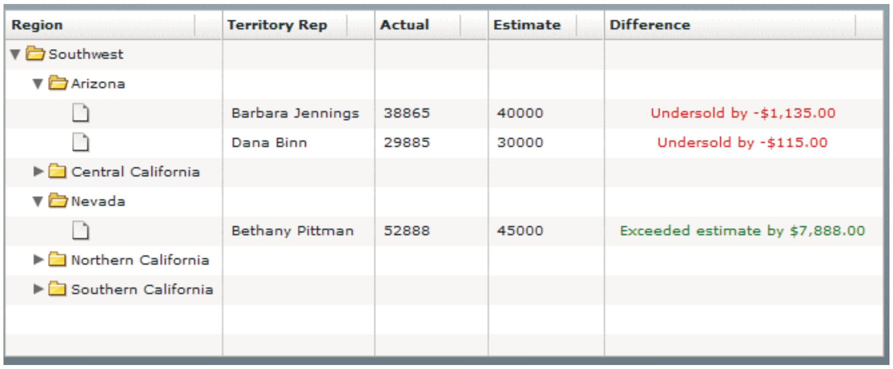
The previous example uses a single renderer that spans the Actual and Estimate columns to display a combined view of the data for these columns. For an implementation of the SummaryRenderer component, see “Using a renderer to generate column data” on page 1445.

The following table describes the properties of the AdvancedDataGridRendererProvider class that you use to configure the renderer:

Property	Description
`column`	The ID of the column for which the renderer is used. If you omit this property, you can use the `columnIndex` property to specify the column.
`columnIndex`	The column index for which the renderer is used, where the first column has an index of 0.
`columnSpan`	The number of columns that the renderer spans. Set this property to 0 to span all columns. The default value is 1.
`dataField`	The data field in the data provider for the renderer. This property is optional.
`depth`	The depth in the tree at which the renderer is used, where the top-most node of the tree has a depth of 1. Use this property when the renderer should be used only when the tree is expanded to a certain depth, not for all nodes in the tree. By default, the control uses the renderer for all levels of the tree.
`renderer`	The renderer.
`rowSpan`	The number of rows that the renderer spans. The default value is 1.

Using a renderer to generate column data

The following example shows the results when an item renderer is used to calculate the value of the Difference column of the AdvancedDataGrid control:



Region	Territory Rep	Actual	Estimate	Difference
▼ Southwest				
▼ Arizona				
	Barbara Jennings	38865	40000	Undersold by -$1,135.00
	Dana Binn	29885	30000	Undersold by -$115.00
▶ Central California				
▼ Nevada				
	Bethany Pittman	52888	45000	Exceeded estimate by $7,888.00
▶ Northern California				
▶ Southern California				

This example defines a column with the `id` of diffCol that is not associated with a data field in the data provider. Instead, you use the `rendererProvider` property to assign an item renderer to the column. The item renderer calculates the difference between the actual and estimate values, and then displays a message based on whether the representative exceeded or missed their estimate.

The following code implements this example:

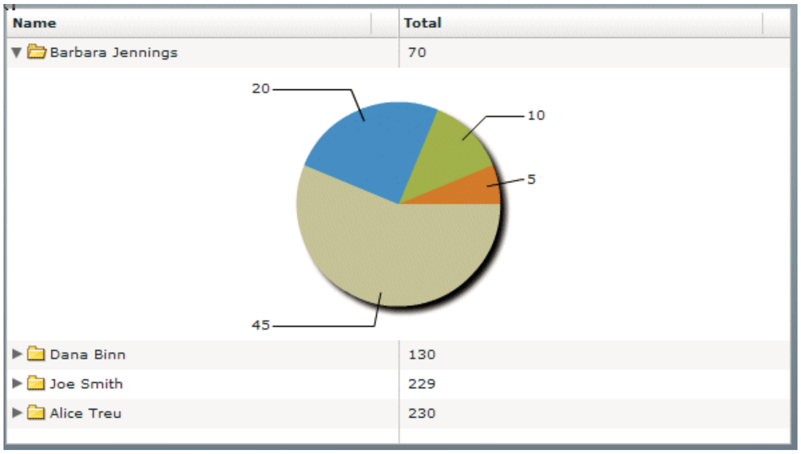
```
<?xml version="1.0"?>
<!-- dpcontrols/adg/HierarchicalADGSimpleRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="750">
    <fx:Script>
        <![CDATA[
              import mx.collections.ArrayCollection;

              include "SimpleHierarchicalData.as";
        ]]>
    </fx:Script>
    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn id="diffCol"
                headerText="Difference"/>
        </mx:columns>
        <mx:rendererProviders>
            <mx:AdvancedDataGridRendererProvider column="{diffCol}"
                depth="3" renderer="myComponents.SummaryRenderer"/>
        </mx:rendererProviders>
    </mx:AdvancedDataGrid>
</s:Application>
```

The following code shows the SummaryRenderer component:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/myComponents/SummaryRenderer.mxml -->
<mx:Label xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" textAlign="center">
    <fx:Script>
        <![CDATA[
        override public function set data(value:Object):void
        {
            // Calculate the difference.
            var diff:Number =
                Number(value["Actual"]) - Number(value["Estimate"]);
            if (diff < 0)
            {
                // If Estimate was greater than Actual,
                // display results in red.
                setStyle("color", "red");
                text = "Undersold by " + usdFormatter.format(diff);
            }
            else
            {
                // If Estimate was less than Actual,
                // display results in green.
                setStyle("color", "green");
                text = "Exceeded estimate by " + usdFormatter.format(diff);
            }
        }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <s:CurrencyFormatter id="usdFormatter"
            useCurrencySymbol="true" negativeCurrencyFormat="1"/>
    </fx:Declarations>
</mx:Label>
```

Using an item renderer that spans an entire row

You can use an item renderer with a hierarchical data set to display an entire row of data. The following example shows a PieChart control that displays the data from the detail field of the hierarchical data set. Each row contains detailed information about sales revenues for the representative, which is rendered as a segment of the pie chart:



Name	Total
▼ Barbara Jennings	70
▶ Dana Binn	130
▶ Joe Smith	229
▶ Alice Treu	230

The following code implements this example:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/GroupADGChartRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      private var dpHierarchy:ArrayCollection= new ArrayCollection([
        {name:"Barbara Jennings", region: "Arizona", total:70, children:[
          {detail:[{amount:5},{amount:10},{amount:20},{amount:45}]}]},
        {name:"Dana Binn", region: "Arizona", total:130,  children:[
          {detail:[{amount:15},{amount:25},{amount:35},{amount:55}]}]},
        {name:"Joe Smith", region: "California", total:229,  children:[
          {detail:[{amount:26},{amount:32},{amount:73},{amount:123}]}]},
        {name:"Alice Treu", region: "California", total:230, children:[
          {detail:[{amount:159},{amount:235},{amount:135},{amount:155}]}
        ]}
      ]);
    ]]>
  </fx:Script>
  <mx:AdvancedDataGrid id="myADG"
    width="100%" height="100%"
    variableRowHeight="true">
    <mx:dataProvider>
       <mx:HierarchicalData source="{dpHierarchy}"/>
    </mx:dataProvider>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="name" headerText="Name"/>
        <mx:AdvancedDataGridColumn dataField="total" headerText="Total"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            dataField="detail"
            renderer="myComponents.ChartRenderer"
            columnIndex="0"
            columnSpan="0"/>
    </mx:rendererProviders>
  </mx:AdvancedDataGrid>
</s:Application>
```

The following code renders the ChartRenderer.mxml component:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/myComponents/ChartRenderer.mxml -->
<mx:VBox height="200" width="100%" xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:PieChart dataProvider="{data.detail}"
        width="100%"
        height="100%"
        showDataTips="true">
        <mx:series>
            <mx:PieSeries labelPosition="callout" field="amount"/>
        </mx:series>
    </mx:PieChart>
</mx:VBox>
```

You can modify this example to display the PieChart control in a column. In the following example, you add a Detail column to the control, and then specify that the item renderer is for column 2 of the control:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/GroupADGChartRendererOneRow.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      private var dpHierarchy:ArrayCollection= new ArrayCollection([
        {name:"Barbara Jennings", region: "Arizona", total:70, children:[
          {detail:[{amount:5},{amount:10},{amount:20},{amount:45}]}]},
        {name:"Dana Binn", region: "Arizona", total:130,  children:[
          {detail:[{amount:15},{amount:25},{amount:35},{amount:55}]}]},
        {name:"Joe Smith", region: "California", total:229,  children:[
          {detail:[{amount:26},{amount:32},{amount:73},{amount:123}]}]},
        {name:"Alice Treu", region: "California", total:230, children:[
          {detail:[{amount:159},{amount:235},{amount:135},{amount:155}]}
        ]}
      ]);
    ]]>
  </fx:Script>
  <mx:AdvancedDataGrid id="myADG"
    width="100%" height="100%"
    variableRowHeight="true">
    <mx:dataProvider>
       <mx:HierarchicalData source="{dpHierarchy}"/>
    </mx:dataProvider>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="name" headerText="Name"/>
        <mx:AdvancedDataGridColumn dataField="total" headerText="Total"/>
        <mx:AdvancedDataGridColumn dataField="detail" headerText="Detail"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            dataField="detail"
            renderer="myComponents.ChartRenderer"
            columnIndex="2"/>
    </mx:rendererProviders>
  </mx:AdvancedDataGrid>
</s:Application>
```

Keyboard navigation

The following keystrokes are built into the AdvancedDataGrid control to let users navigate the control:

Keyboard action	**Key assignments**
Standard	• Use the Up and Down Arrow keys to scroll vertically by one row. • Type characters to do incremental type-ahead look-ups in the first column. • Use the Home and End keys to move to the first and last rows. • Use the Page Up and Page Down keys to move to the last visible row and then scroll vertically by the number of rows specified by the `rowcount` property.
In editable mode	• Use the Tab and Shift+Tab keys to move to the next and previous cells, and start editing. • Use the Escape or Control+Period keys to cancel editing. • Use the Enter key based on the value of the `AdvancedDataGridColumn.editorUsesEnterKey` property: • If `editorUsesEnterKey=true`, the Enter key is considered to be part of the input text. • If `editorUsesEnterKey=false`, if single-line editor, the Enter key saves the item and moves down a row. • If you use the Tab key to move focus to the AdvancedDataGrid control, focus is placed on the position of the last edited item. • Use the Control click keys to select multiple discontiguous items.
For the expandable tree	• Use the Multiply key on the number pad (asterisk) to open or close all nodes under the current node, with no animation. • Use the Add key on the number pad or the Space key to open nodes with animation. • Use the Subtract key on the number pad to close nodes with animation. • Use the Control+Shift+Space key to open a closed node, or close an open node. • Use the Control+Shift+Right Arrow key to open a closed node, or select the parent node of a leaf node. • Use the Control+Shift+Left Arrow to close open nodes. • Use the Up and Down Arrow keys to move up and down one row.

Keyboard action	**Key assignments**
For column headers	• Use the arrow keys to move focus to a column header. • The first Space key pressed when a column header has focus sorts the column in descending order. Each subsequent press of the Space key toggles the sort order between ascending and descending order. • Use the Control+Space keys when a header has focus to sort multiple columns. • Use the Left Arrow and Right Arrow keys to move to the previous or next header, without column wrapping.
To scroll vertically and horizontally in pages	• The first Page Up or Page Down key pressed move to the first or last visible row. The next Page Up or Page Down key pressed scrolls vertically up or down by a page, which corresponds to the number of visible rows. • Use the Shift+Page Up and Shift+Page Down keys to move to first or last visible column.

Keyboard action	Key assignments
To move focus out of AdvancedDataGrid while editing is in progress	When you set `AdvancedDataGrid.editable=true` or `AdvancedDataGridColumn=true`: • Use the Tab key to move focus to the AdvancedDataGrid control. • Use the Tab key again to move focus to the next component. • Use the Left, Right, Up, and Down Arrow keys to move between cells. • Use the Shift+Home and Shift+End keys to move to the first and last column in current row. • Cells are only selected by default, they are not editable. • Press the F2 key to make a cell editable. • Use the arrow keys to move the cursor in the editing area. • Use the Escape key to cancel editing. • Use the Enter key to commit editing changes. • Same behavior as when `editable=true`, but the F2 key does not make the cell editable. • Use the Home and End keys to move to first and last rows. • Use the Page Up and Page Down keys to move to the previous and next pages. When focus is on the AdvancedDataGrid control: • Use the Left, Right, Up, and Down Arrow keys to move between cells. • Use the Shift+Home and Shift+End keys to move to the first and last column in current row. • Cells are only selected by default, they are not editable. • Press the F2 key to make a cell editable. • Use the arrow keys to move the cursor in the editing area. • Use the Escape key to cancel editing. • Use the Enter key to commit editing changes. • Same behavior as when `editable=true`, but the F2 key does not make the cell editable. • Use the Home and End keys to move to first and last rows. • Use the Page Up and Page Down keys to move to the previous and next pages. When editing the cell: • Use the arrow keys to move the cursor in the editing area. • Use the Escape key to cancel editing. • Use the Enter key to commit editing changes. • Same behavior as when `editable=true`, but the F2 key does not make the cell editable. • Use the Home and End keys to move to first and last rows. • Use the Page Up and Page Down keys to move to the previous and next pages. When you set `AdvancedDataGrid.editable=false` or `AdvancedDataGridColumn=false`: • Same behavior as when `editable=true`, but the F2 key does not make the cell editable. • Use the Home and End keys to move to first and last rows. • Use the Page Up and Page Down keys to move to the previous and next pages.

OLAPDataGrid control

The OLAPDataGrid control lets you aggregate large amounts of data for easy display and interpretation. The control supports a two-dimensional grid layout, where the columns and rows of the control can display a single level of information, or you can create hierarchical rows and columns to display more complex data.

The OLAPDataGrid controls is a subclass of the AdvancedDataGrid control, and therefore supports many of the features of the AdvancedDataGrid control. For more information on the AdvancedDataGrid control, see "AdvancedDataGrid control" on page 1397.

About OLAP data grids

When working with large amounts of data, you can quickly get overwhelmed with the scope and size of the data. For example, you collect sales information for different products, in different regions, and for different customers in a typical two-dimensional spreadsheet. That spreadsheet could easily contain hundreds of rows and tens or even hundreds of columns. Extracting useful information for such a large data collection can be difficult, and trying to identify trends or other patterns in the data can be even harder.

Data visualization is a technique for examining large amounts of data in a compact format. One type of data visualization technique is to use a chart, such as a bar, column, or pie chart. Adobe® Flex™ supports many types of charts. For more information, see "Introduction to charts" on page 1075.

Another data visualization technique is to aggregate the data in a compact format, such as in an OLAP (online analytical processing) data grid. An OLAP data grid is similar to a pivot table in Microsoft Excel. An OLAP data grid displays data aggregations in a two-dimensional grid of rows and columns, like a spreadsheet, but the data is condensed based on your aggregation settings.

***Note:** While the Flex OLAP data grid is similar to a pivot table, it provides a much greater set of features for aggregating data.*

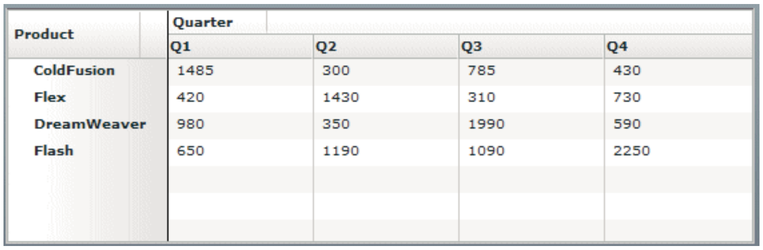
For example, you collect sales information on a server in a flat data structure of records, where each record contains information for a single customer transaction, for a single product, in a single quarter. The following code shows the format of this flat data:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    year:"2007",
    quarter:"Q1"
    revenue: "100.00"
}
```

For a large company with hundreds of customers and tens or hundreds of products, this table could easily contain several thousand rows. Rather than display this information in a standard spreadsheet, you download your data to a Flex application to aggregate sales data by product and quarter, and then display the aggregated data in an OLAP data grid. From this data aggregation, you can determine trends in sales of each product over time.

About the OLAPDataGrid control

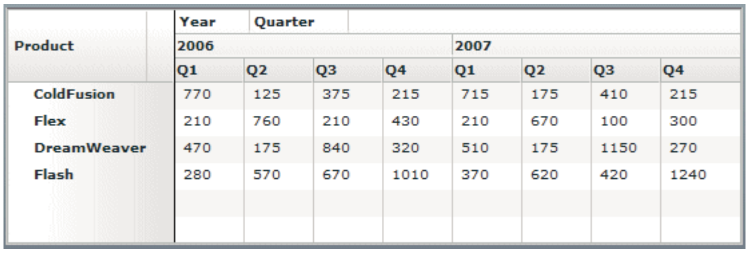
You use the Flex OLAPDataGrid control to display an OLAP data grid. The following image shows the OLAPDataGrid control displaying the aggregated sales information for product and quarter:



Product	Quarter			
	Q1	Q2	Q3	Q4
ColdFusion	1485	300	785	430
Flex	420	1430	310	730
DreamWeaver	980	350	1990	590
Flash	650	1190	1090	2250

Like all Flex data grid controls, the OLAPDataGrid control is designed to display data in a two-dimensional representation of rows and columns.

You can modify this example to compare quarterly sales from two different years, as the following example shows:

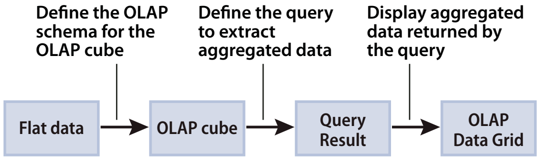


Product	Year	Quarter						
	2006				2007			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
ColdFusion	770	125	375	215	715	175	410	215
Flex	210	760	210	430	210	670	100	300
DreamWeaver	470	175	840	320	510	175	1150	270
Flash	280	570	670	1010	370	620	420	1240

In the previous figure, the columns of the OLAPDataGrid control show a hierarchy of information for year and quarter. You can add multiple-level hierarchies for both the columns and the rows of the control.

About creating an OLAP data grid

The following figure shows the data flow that you use to aggregate your data for display in the OLAPDataGrid control:


Define the OLAP schema for the OLAP cube
Define the query to extract aggregated data
Display aggregated data returned by the query
Flat data
OLAP cube
Query Result
OLAP Data Grid

The following steps describe this process in more detail:

1 Start with flat data, which is typically data arranged as a set of records where each record contains the same data fields. For example, you might start with flat data from a spreadsheet, or with data from a table of a relational database.

  The following code shows an example format of flat data:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    quarter:"Q1"
    revenue: "100.00"
}
```

2 Define an OLAP schema that describes how your data gets transformed from a flat representation into an OLAP cube. An OLAP schema defines the representation of your flat data in an OLAP cube, and defines how to aggregate your data for an OLAP query.

An OLAP cube is analogous to a table in a relational database. But whereas a table typically has two dimensions (row and column), an OLAP cube can have any number of dimensions. In this example, the cube has three dimensions: customer, product, and quarter. Every possible set of values for customer, product, and quarter defines a unique point in the cube. The value at each point in the cube is the sales revenue for that set of values of customer, product, and quarter.

3 Create queries to extract aggregated data from the OLAP cube for display in the OLAPDataGrid control.

After your data is in an OLAP cube, you write queries to extract aggregated data for display in the OLAPDataGrid control. You can write multiple queries to create different types of data aggregations.

4 Use the query results as input to the OLAPDataGrid control to display the results.

About OLAP cubes

An OLAP cube can have any number of dimensions. In its simplest form, the *dimensions* of an OLAP cube correspond to a field of the flat data set. For example, you have flat data that contains three fields:

```
data:Object = {
    product:"ColdFusion"
    quarter :"Q1"
    revenue: "100.00",
}
```

The data fields of each record can contain the following values:

- The product field can have the values: ColdFusion, Flex, Dreamweaver, and Illustrator.
- The quarter field can have the values: Q1, Q2, Q3, and Q4.
- The revenue field contains the sales, in dollars, of the product for the quarter.

To aggregate your data, you create an OLAP cube with two dimensions: quarter and product. The value along each dimension of the cube is called a *member*. For example, the product dimension of the cube has the following members: ColdFusion, Flex, Dreamweaver, and Illustrator. For the quarter dimension, the members are Q1, Q2, Q3, and Q4.

The value at any point in the cube defined by the two dimensions is called a *measure* of the cube. For example, the measure at the point in the cube defined by (Q1, ColdFusion) is 100.00. A schema can define one or more measures for a single point in the OLAP cube.

Flex supports only numeric values for the measure of a cube. The advantage of numeric values is that they can be easily aggregated for display in the OLAPDataGrid control. Some typical aggregation types include sum, average, minimum, and maximum. For example, you specify the aggregation method of the revenue measure as `SUM`. You then extract sales information from the cube for ColdFusion. The aggregated sales data contains the sum of all ColdFusion sales for each quarter.

About OLAP schemas

To convert flat data into an OLAP cube, you create an OLAP *schema* that defines the dimensions of the cube, the fields of the flat data that supply the members along each dimension, and the fields of the flat data that supply the measure for any point in the cube.

For example, you have the following flat data that contains sales records:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    quarter:"Q1"
    revenue: "100.00"
}
```

The following example shows the definition for an OLAPCube that includes the definition of the OLAP schema used to represent this data in the cube. This schema defines a three-dimensional OLAP cube based on the customer, product, and quarter fields of the data.

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="CustomerDim">
        <mx:OLAPAttribute name="Customer" dataField="customer"/>
        <mx:OLAPHierarchy name="CustomerHier" hasAll="true">
            <mx:OLAPLevel attributeName="Customer"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>
    <mx:OLAPDimension name="QuarterDim">
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue"
        aggregator="SUM"/>
</mx:OLAPCube>
```

Notice that in this schema:

- All dimensions are defined first, and then all measures.
- The first line of each dimension associates a data field of the flat data with an OLAPAttribute instance. You then use the `OLAPLevel.attributeName` property to associate the attribute with a level of the dimension to populate the members of the dimension. For example, in this schema you populate the Customer level of the CustomerDim dimension with the data from the customer field of the data.
- A dimension of an OLAP schema always contains a hierarchy of one or more levels. In this schema, the hierarchy of each dimension contains only a single level corresponding to a field of the flat data. This is the simplest form of a dimension. Other schemas could define multiple levels in a hierarchy to create a complex dimension. For more information, see "Creating an OLAP schema" on page 1462.

- A measure definition specifies the data field of the flat data that contains the value for each point in the OLAP cube, and how the measure is aggregated by an OLAP query. In this example, you aggregate revenue by summing it. That means all queries of this OLAP cube will return revenue summations. Other types of aggregation methods include maximum, minimum, and average.
- This definition of the OLAPCube specifies an event handler for the `complete` event. You cannot invoke a query on the cube until it completes initialization, which is signalled by the cube when it dispatches the `complete` event.

Based on the requirements of your application, you might create multiple OLAP cubes from the same flat data set, where each cube uses a different schema to create its own arrangement of dimensions and measures. For more information and examples of OLAP schemas, see "Creating an OLAP schema" on page 1462.

About OLAP queries

OLAP queries extract aggregated data from an OLAP cube for display in an OLAPDataGrid control. The query specifies the dimensions that define the characteristics of the query, and the measure or measures aggregated to create the query results.

In the previous section, you defined an OLAP schema for sales information where the schema defines CustomerDim, ProductDim, and QuarterDim dimensions, and a single measure for revenue aggregated by the `SUM` aggregation method. Therefore, you can create a query to sum revenue by the following criteria:

- Product for each quarter
- Customer for each product
- Customer for each quarter
- Any other combination of members from each dimension

You construct a query in ActionScript as an instance of the OLAPQuery class, and then execute the query by calling the `OLAPCube.execute()` method, which returns an instance of the AsyncToken class.

A query is required to have two axes, a row axis and a column axis, of type IOLAPQueryAxis. The row axis defines the data aggregation information for each row of the OLAPDataGrid control, and the column axis defines the data aggregation information for each column of the control.

You use the OLAPSet class to specify the data aggregation information for each axis, as the following example shows:

```
// Create an instance of OLAPQuery to represent the query.
var query:OLAPQuery = new OLAPQuery;

// Get the row axis from the query instance.
var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
// Create an OLAPSet instance to configure the axis.
var productSet:OLAPSet = new OLAPSet;
// Add the Product to the row to aggregate data
// by the Product dimension.
productSet.addElements(
    cube.findDimension("ProductDim").findAttribute("Product").children);
// Add the OLAPSet instance to the axis.
rowQueryAxis.addSet(productSet);

// Get the column axis from the query instance, and configure it
// to aggregate the columns by the Quarter dimension.
var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
var quarterSet:OLAPSet= new OLAPSet;
quarterSet.addElements(
    cube.findDimension("QuarterDim").findAttribute("Quarter").children);
colQueryAxis.addSet(quarterSet);

// Execute the query.
var token:AsyncToken = cube.execute(query);
// Set up handlers for the query results.
token.addResponder(new AsyncResponder(showResult, showFault));
```

Notice that in this query:

- You initialize each axis by calling the `OLAPQuery.getAxis()` method.
- You configure each OLAPSet instance by calling the `OLAPSet.addElements()` method to specify the information used to populate the axis.
- You set up two functions to handle the query result defined by the AsyncToken class. In this example, the function `showResult()` handles the query results when the query succeeds, and the function `showFault()` handles any errors detected during query execution. For more information on using the AsyncToken class, see “Executing a query and returning the results to an OLAPDataGrid control” on page 1469.

For more information and examples of OLAP queries, see “Creating OLAP queries” on page 1466.

The differences between the OLAPDataGrid and the AdvancedDataGrid control

You use the OLAPDataGrid control to display the results of an OLAP query. The OLAPDataGrid control is a subclass of the AdvancedDataGrid control and inherits much of its functionality. However, because of the way you pass data to the OLAPDataGrid control, it has several differences from the AdvancedDataGrid control:

- Column dragging is not allowed in the OLAPDataGrid control.
- You cannot edit cells in the OLAPDataGrid control because cell data is a result of a query and does not correspond to a single data value in the OLAP cube.
- You cannot sort columns by clicking on headers in the OLAPDataGrid control. Sorting is supported at the dimension level so that you can change the order of members of that dimension. Note that you can only use sorting while preparing the query, and then use it to get the member names displayed along the rows and column headers.

You populate an OLAPDataGrid control with data by setting its data provider to an instance of the OLAPResult class, which contains the results of an OLAP query. For a complete example that uses this control, see “Example using the OLAPDataGrid control” on page 1460.

Example using the OLAPDataGrid control

The following example shows the complete code for creating the OLAPDataGrid control that appears in the section "About OLAP data grids" on page 1454. This example aggregates product sales by quarter. The example uses flat data from the dataIntro.as file, which contains sales records in the following format:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    quarter:"Q1"
    revenue: "100.00"
}
```

The following code implements this example:

```
<?xml version="1.0"?>
<!-- olapdatagrid/OLAPDG_Intro.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
        creationComplete="creationCompleteHandler();">
    <fx:Script>
      <![CDATA[
        import mx.rpc.AsyncResponder;
        import mx.rpc.AsyncToken;
        import mx.rpc.events.FaultEvent;
        import mx.olap.OLAPQuery;
        import mx.olap.OLAPSet;
        import mx.olap.IOLAPQuery;
        import mx.olap.IOLAPQueryAxis;
        import mx.olap.IOLAPCube;
        import mx.olap.OLAPResult;
        import mx.events.CubeEvent;
        import mx.controls.Alert;
        import mx.collections.ArrayCollection;

        include "dataIntro.as"
        private function creationCompleteHandler():void {
            // You must initialize the cube before you
            // can execute a query on it.
            myMXMLCube.refresh();
        }
        // Create the OLAP query.
        private function getQuery(cube:IOLAPCube):IOLAPQuery {
            // Create an instance of OLAPQuery to represent the query.
            var query:OLAPQuery = new OLAPQuery;

            // Get the row axis from the query instance.
            var rowQueryAxis:IOLAPQueryAxis =
                query.getAxis(OLAPQuery.ROW_AXIS);
            // Create an OLAPSet instance to configure the axis.
            var productSet:OLAPSet = new OLAPSet;
            // Add the Product to the row to aggregate data
            // by the Product dimension.
            productSet.addElements(
                cube.findDimension("ProductDim").findAttribute("Product").children);
            // Add the OLAPSet instance to the axis.
            rowQueryAxis.addSet(productSet);
            // Get the column axis from the query instance, and configure it
            // to aggregate the columns by the Quarter dimension.
            var colQueryAxis:IOLAPQueryAxis =
                query.getAxis(OLAPQuery.COLUMN_AXIS);
            var quarterSet:OLAPSet= new OLAPSet;
            quarterSet.addElements(
                cube.findDimension("QuarterDim").findAttribute("Quarter").children);
            colQueryAxis.addSet(quarterSet);

            return query;
        }
        // Event handler to execute the OLAP query
        // after the cube completes initialization.
        private function runQuery(event:CubeEvent):void {
            // Get cube.
            var cube:IOLAPCube = IOLAPCube(event.currentTarget);
            // Create a query instance.
            var query:IOLAPQuery = getQuery(cube);
            // Execute the query.
            var token:AsyncToken = cube.execute(query);
            // Set up handlers for the query results.
            token.addResponder(new AsyncResponder(showResult, showFault));
        }
        // Handle a query fault.
        private function showFault(error:FaultEvent, token:Object):void {
            Alert.show(error.fault.faultString);
        }
        // Handle a successful query by passing the query results to
        // the OLAPDataGrid control..
        private function showResult(result:Object, token:Object):void {
            if (!result) {
                Alert.show("No results from query.");
                return;
            }
            myOLAPDG.dataProvider= result as OLAPResult;
        }
      ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:OLAPCube name="FlatSchemaCube"
            dataProvider="{flatData}"
            id="myMXMLCube"
            complete="runQuery(event);">

            <mx:OLAPDimension name="CustomerDim">
                <mx:OLAPAttribute name="Customer" dataField="customer"/>
                <mx:OLAPHierarchy name="CustomerHier" hasAll="true">
                    <mx:OLAPLevel attributeName="Customer"/>
                </mx:OLAPHierarchy>
            </mx:OLAPDimension>
            <mx:OLAPDimension name="ProductDim">
                <mx:OLAPAttribute name="Product" dataField="product"/>
                <mx:OLAPHierarchy name="ProductHier" hasAll="true">
                    <mx:OLAPLevel attributeName="Product"/>
                </mx:OLAPHierarchy>
            </mx:OLAPDimension>
            <mx:OLAPDimension name="QuarterDim">
                <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
                <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
                    <mx:OLAPLevel attributeName="Quarter"/>
                </mx:OLAPHierarchy>
            </mx:OLAPDimension>

            <mx:OLAPMeasure name="Revenue"
                dataField="revenue"
                aggregator="SUM"/>
        </mx:OLAPCube>
    </fx:Declarations>
    <mx:OLAPDataGrid id="myOLAPDG" width="100%" height="100%"/>
</s:Application>
```

Creating an OLAP schema

An OLAP schema defines the representation of your flat data in an OLAP cube, and defines how to aggregate your data for an OLAP query. You typically define your schema in MXML. While you can construct a schema programmatically in ActionScript, that method requires much more coding than MXML.

The following table describes the classes and interfaces that you use to define an OLAP schema:

Class	Interface	Description
OLAPSchema	IOLAPSchema	The schema instance.
OLAPCube	IOLAPCube	The OLAP cube created by the schema.
OLAPDimension	IOLAPDimension	A dimension of the schema.
OLAPAttribute	IOLAPAttribute	An attribute of a dimension
OLAPHierarchy	IOLAPHierarchy	A hierarchy of a dimension.
OLAPLevel	IOLAPLevel	A level of a hierarchy.
OLAPMeasure	IOLAPMeasure	A measure of a dimension.

General form of a schema definition

In MXML, you can define an OLAP schema as part of the definition of an OLAPCube instance. In MXML, you declare an OLAPCube in an `<fx:Declarations>` tag because it is not a visual component.

Typically, you set any properties of the OLAPCube instance as tag attributes, and set the `OLAPCube.dimensions` and `OLAPCube.measures` properties as child tags, as the following example shows:

```
<fx:Declarations>
    <mx:OLAPCube name="FlatSchemaCube"
        dataProvider="{flatData}"
        id="myMXMLCube"
        complete="runQuery(event);">

        <!-- Define dimensions. -->
        <mx:OLAPDimension ... />

        <mx:OLAPDimension ... />
        ...

        <!-- Define measures. -->
        <mx:OLAPMeasure ... />
        ...
    </mx:OLAPCube>
</fx:Declarations>
```

The order of the OLAPDimension and OLAPMeasure definitions is not important, but you should not place OLAPMeasure definitions between OLAPDimension definitions.

Specifying the aggregation method for a measure

As part of creating an OLAP schema, you specify the data field that provides the value of the measure for each point in the cube. The measure corresponds to the data value at that point in the OLAP cube. For example, if you define a schema for sales information, you might specify as a measure of the cube the revenue for a product, and the aggregation type as `SUM`.

A schema can define one or more measures for a single point in the OLAP cube. The first measure in the schema is called the *default measure*, and is the measure returned by an OLAP query when you do not explicitly specify the measure to return. For more information on selecting a specific measure, see “Creating a query using a nondefault measure” on page 1484.

The following example shows a section of an MXML schema definition that specifies two measures for the schema:

```
<mx:OLAPMeasure name="Revenue" dataField="revenue" aggregator="SUM"/>
<mx:OLAPMeasure name="Cost" dataField="cost" aggregator="SUM"/>
```

When creating a schema, you specify the name of the measure, the data field in the input flat data that contains the data for the measure, and an aggregation method of the measure. In this example, the aggregation method is `SUM`. That means when you create an OLAP query for a measure, the query sums all revenue fields to generate the values displayed by the OLAPDataGrid control. You could use this schema to define a cube so that you can total sales and cost information for products, regions, and other characteristics of your data.

To aggregate the revenue data using a different aggregation method, such as average or maximum, create another schema to define a second OLAP cube. The following example shows a section of another MXML schema definition that specifies the aggregation method as `MAX` for the sales data and `MIN` for the cost data:

```
<mx:OLAPMeasure name="Revenue" dataField="revenue" aggregator="MAX"/>
<mx:OLAPMeasure name="Cost" dataField="cost" aggregator="MIN"/>
```

You should take care in how you define your schema because it can limit the types of queries that you can run on it, or the level of detail at which you can create aggregations.

Creating a schema dimension

An OLAPSchema instance can define any number of dimensions, limited only by your input data. A dimension can be a simple dimension with a single level, or it can be a complex dimension with multiple levels. The dimension of a schema always has the same basic form:

```
<mx:OLAPDimension name="QuarterDim">
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    ...
    <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
        <mx:OLAPLevel attributeName="Quarter"/>
        ...
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

Notice that in this schema:

- The dimension first uses OLAPAttribute class to specify the data fields of the input data set used to populate the members of the dimension.

  The dimension defines an instance of the OLAPHierarchy class. Therefore, a dimension is always assumed to contain a hierarchy of levels, even if that hierarchy contains only a single level.

- The dimension specifies one or more instances of the OLAPLevel class to associate a field of the input with the dimension.

Defining a dimension that contains a single level

To create a simple dimension (a dimension that contains a single level), you define a dimension hierarchy, as the following example shows:

```
<mx:OLAPDimension name="CustomerDim">
    <mx:OLAPAttribute name="Customer" dataField="customer"/>
    <mx:OLAPHierarchy name="CustomerHier" hasAll="true">
        <mx:OLAPLevel attributeName="Customer"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

In this example, the customer field of the input data defines the entire measure of the dimension.

Defining a dimension that contains multiple levels

Instead of creating a separate dimension for each field of your data, you can choose to group related data fields along a single dimension. For example, your data might contain several fields related to the time of a transaction, such as month, quarter, and year. Or your data might contain multiple fields associated with the geographical area of a transaction, such as region, state, province, or country.

The TimeDim dimension in the following example defines hierarchical dimensions that contain two levels, one for year and one for quarter:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube">

    <mx:OLAPDimension name="TimeDim">
        <mx:OLAPAttribute name="Year" dataField="year"/>
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
            <mx:OLAPLevel attributeName="Year"/>
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"dataField="revenue" aggregator="SUM"/>
</mx:OLAPCube>
```

With this schema definition, you can aggregate data for all time, for individual years, for individual quarters, or for individual quarters of a specific year.

Notice that the TimeDim dimension contains a Year level, the most general level, and a Quarter level, the more detailed level. The first level in the hierarchy typically defines the most general level, and each subsequent level provides a greater level of detail.

If your data contained a month field with the month of the year for a transaction, you can add the Month level to the TimeDim dimension. Since month is a more detailed measure of time than quarter, add the Month level after the Quarter level, as the following example shows:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPAttribute name="Month" dataField="month"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
        <mx:OLAPLevel attributeName="Month"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

The advantage of creating a hierarchical schema is that you can write queries to extract top-level data aggregations or to drill down into the data to extract more granular data. For the previous schema definition, you can write an OLAP query to aggregate data by the most general field, such as year, or drill down to obtain more granular results by aggregating your data by quarter in each year, by month in each quarter, or by month in each year.

Creating a default member in a schema

Most of the OLAP schemas shown so far have been defined in the following form:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

In this schema, the hierarchy sets the `OLAPHierarchy.hasAll` property to `true` to create a default member for the hierarchy. The default member is created automatically for the hierarchy and contains an aggregation of all levels in the hierarchy. In the previous schema, the default member contains an aggregation of the measure of the schema for all years and all quarters.

The default member is used by OLAP queries when you do not specify any criteria to aggregate the dimension. For example, your OLAP schema contains a ProductDim, TimeDim, and CustomerDim dimension. You then write a query to extract data by product and customer, but omit any specification in the query for time. The OLAP query automatically uses the default member of the dimension to aggregate the information for the TimeDim dimension.

The default value of the `OLAPHierarchy.hasAll` property is `true`. If you set it to `false`, the OLAP query uses the first level in the hierarchy to aggregate the dimension. The following schema sets the `hasAll` property to `false`:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="false">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

Because the `hasAll` property is `false`, an OLAP query automatically aggregates the data in the dimension by the Year level when you omit the TimeDim dimension from the query.

By default, the OLAP cube adds a member to the dimension named `(All)` to represent the default member. You can use the `OLAPHierarchy.allLevelName` property to specify a different name, as the following example shows:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year" allLevelname="AllTime"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

When you execute an OLAP query, you choose whether or not to include the default member in the query results. For more information, see "Using the default member in a query" on page 1471.

Creating OLAP queries

You use an OLAP query to extract data from an OLAP cube for display by the OLAPDataGrid control. An OLAP query defines a result set in a two-dimensional table of rows and columns so that the data can be viewed in the OLAPDataGrid control.

The following table describes the classes and interfaces that you use the define a query:

Class	Interface	Description
AsyncToken		The result of the `OLAPQuery.execute()` method.
OLAPCell	IOLAPCell	An area of the cube defined by a tuple.
OLAPMember	IOLAPMember	A member of a dimension.
OLAPQuery	IOLAPQuery	The query instances.
OLAPQueryAxis	IOLAPQueryAxis	An axis of the query.

Class	Interface	Description
OLAPResult	IOLAPResult	The query result.
OLAPSet		A set of members for an axis.
OLAPTuple	IOLAPTuple	A tuple containing one or more members.

***Note:** Many of the examples in this topic use interfaces rather than classes. One of the reasons to use interfaces is that if you define customized versions of classes that implement these interfaces, you do not have to modify your code.*

Creating a query

The process of creating a query has the following steps:

1 Prepare the cube for a query. For more information, see "Preparing a cube for a query" on page 1467.

2 Define an instance of the OLAPQuery class to represent the query. For more information, see "Creating a query axis" on page 1468.

3 Define an instance of the OLAPQueryAxis class to represent the rows of the query.

4 Define an instance of the OLAPSet class to define the members that provide the row axis information. For more information, see "Writing a query for a simple OLAP cube" on page 1471 and "Writing a query for a complex OLAP cube" on page 1478.

5 Define an instance of the OLAPQueryAxis class to represent the columns of the query.

6 Define an instance of the OLAPSet class to define the members that provide the axis column information.

7 Optionally define an instance of the OLAPQueryAxis class to specify a slicer axis. For more information, see "Creating a slicer axis" on page 1484

8 Define an instance of the OLAPSet class to define the members that provide the slicer axis information.

9 Execute the query on the cube by calling `OLAPCube.execute()`. For more information, see "Executing a query and returning the results to an OLAPDataGrid control" on page 1469.

10 Pass the query results to the OLAPDataGrid control.

Preparing a cube for a query

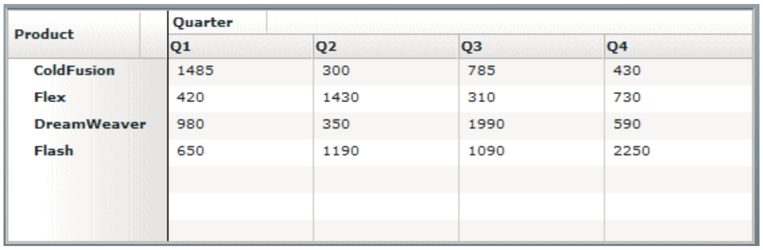
Before you can run the first query on an OLAP cube, you must call the `OLAPCube.refresh()` method to initialize the cube from the input data. Upon completion of cube initialization, the OLAP cube dispatches the `complete` event to signal that the cube is ready for you to query.

You can use event handlers to initialize the cube and to invoke the query. The following example uses the application's `creationComplete` event to initialize the cube, and then uses the cube's `complete` event to execute the query:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="creationCompleteHandler();">

<fx:Script>
    <![CDATA[

        private function creationCompleteHandler():void {
            // You must initialize the cube before you
            // can execute a query on it.
            myMXMLCube.refresh();
        }

        // Event handler to execute the OLAP query
        // after the cube completes initialization.
        private function runQuery(event:CubeEvent):void {
            ...
        }
    ]]>
</fx:Script>
<fx:Dclarations>
    <mx:OLAPCube name="FlatSchemaCube"
        dataProvider="{flatData}"
        id="myMXMLCube"
        complete="runQuery(event);">
        ...
    </mx:OLAPCube>
</fx:Declarations>

...
</s:Application>
```

Creating a query axis

The following example shows an OLAPDataGrid control containing the results of a query for the sales of different products for different quarters:



Product	Quarter			
	Q1	Q2	Q3	Q4
ColdFusion	1485	300	785	430
Flex	420	1430	310	730
DreamWeaver	980	350	1990	590
Flash	650	1190	1090	2250

To create a query, you must define a row axis, a column axis, and optionally a slicer axis, where each axis is an instance of the IOLAPQueryAxis interface. The following table describes the different types of axes:

Axis	Description
Row	Defines the data that appears in each row of the OLAPDataGrid control. In the previous image, you define the row to show each product. This axis is required.
Column	Defines the data displayed in the column. In the previous image, you define the columns to show each year. This axis is required.
Slicer	Optionally defines a filter to reduce the size of the query results, often to reduce the dimensionality of the results from a dimension greater than two so that you can display the results in an OLAPDataGrid control. You also use a slicer access to return data aggregations for the nondefault measure. For more information on using a slicer axis, see "Creating a slicer axis" on page 1484.

When you construct the OLAPQueryAxis instance, you use the `OLAPQuery.getAxis()` method to initialize the axis to configure it as either a row, column, or slicer axis, as the following example shows:

```
// Create an instance of OLAPQuery to represent the query.
var query:OLAPQuery = new OLAPQuery;

// Get the row axis from the query instance.
var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
// Create an OLAPSet instance to configure the axis.
...

var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
...
```

Populating a query axis

You use the OLAPSet class to specify the members of the dimensions of the OLAP cube that populate a query axis. To add a member to an OLAPSet instance, you call the OLAPSet.addElement() or OLAPSet.addElements() method. The following table describes these methods:

OLAPSet method	Description
`addElement(e:IOLAPElement):void`	Adds a single element to the set. • If you pass an IOLAPHierarchy or IOLAPAttributeHierarchy instance to the method, it adds the default member of the hierarchy to the set. • If you pass an IOLAPLevel instance to the method, it adds all the members of the level to the set. • If you pass an IOLAPMember instance, it adds the member to the set.
`addElements(members:IList):void`	Adds multiple elements to the set.

For more information, see "Writing a query for a simple OLAP cube" on page 1471 and "Writing a query for a complex OLAP cube" on page 1478.

Executing a query and returning the results to an OLAPDataGrid control

OLAP cubes can be complex, so you do not want your application to pause while Flex calculates the results of an OLAP query. Therefore, when you execute a query, you also set up two callback functions that handle the results of the query. Flex then calls these functions when the query completes. This architecture lets the query run asynchronously so that your application can continue to execute during query processing.

You execute a query by calling the `OLAPCube.execute()` method, where the `OLAPCube.execute()` method returns an instance of the AsyncToken class. You use the AsyncToken class, along with the AsyncResponder class, to specify the two callback functions to handle the query results when execution completes.

In this example, the function `showResult()` handles the query results when the query succeeds, and the function `showFault()` handles any errors detected during query execution:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="creationCompleteHandler();">

<fx:Script>
    <![CDATA[

        private function creationCompleteHandler():void {
            // You must initialize the cube before you
            // can execute a query on it.
            myMXMLCube.refresh();
        }

        // Create the OLAP query.
        private function getQuery(cube:IOLAPCube):IOLAPQuery {
            ...
        }

        // Event handler to execute the OLAP query
        // after the cube completes initialization.
        private function runQuery(event:CubeEvent):void {
            // Get cube.
            var cube:IOLAPCube = IOLAPCube(event.currentTarget);
            // Create a query instance.
            var query:IOLAPQuery = getQuery(cube);
            // Execute the query.
            var token:AsyncToken = cube.execute(query);
            // Set up handlers for the query results.
            token.addResponder(new AsyncResponder(showResult, showFault));
        }

        // Handle a query fault.
        private function showFault(error:ErrorMessage, token:Object):void {
            Alert.show(error.faultString);
        }

        // Handle a query success.
        private function showResult(result:Object, token:Object):void {
            if (!result) {
                Alert.show("No results from query.");
                return;
            }

            myOLAPDG.dataProvider= result as OLAPResult;
        }
    ]]>
</fx:Script>

<fx:Declarations>
    <mx:OLAPCube name="FlatSchemaCube"
        dataProvider="{flatData}"
        id="myMXMLCube"
        complete="runQuery(event);">
        ...
    </mx:OLAPCube>
<fx:Declarations>

<mx:OLAPDataGrid id="myOLAPDG" width="100%" height="100%" />

</s:Application>
```

Using the default member in a query

Many of the queries shown so far have referenced only two dimensions of the OLAP cube, such as ProductDim and TimeDim. But what happens to the other dimensions when you omit them from the query?

All dimensions have a default member, either explicit or implicit. When you execute a query that does not reference a dimension, the query aggregates the data for that dimension using the default member. For information on defining the default member of a schema, see "Creating a default member in a schema" on page 1465.

Specifying a default cell string for the OLAPDataGrid control

When you define your query, you might create a situation where a cell of the OLAPDataGrid does not contain a value. For example, when aggregating data by customer and product, you might have a customer that has never purchased Illustrator. For that customer and product combination, the corresponding cell of the OLAPDataGrid displays the String "NaN".

You can customize this String by setting the `OLAPDataGrid.defaultCellString`. For example, if you want to set the String to "No value", you would create an OLAPDataGrid control as shown here:

```
<mx:OLAPDataGrid id="myOLAPDG"
    defaultCellString="No value"
    width="100%" height="100%"/>
```

Writing a query for a simple OLAP cube

In a simple OLAP cube, all of the dimensions of the cube define a single level. For example, you have flat data that contains three fields:

```
data:Object = {
    product:"ColdFusion"
    quarter :"Q1"
    revenue: "100.00",
}
```

The data fields of each record can contain the following values:

- The product field can have the values: ColdFusion, Flex, Dreamweaver, and Illustrator.

- The quarter field can have the values: Q1, Q2, Q3, and Q4.

The OLAP schema for this data defines two dimensions, each with a single level, as the following code shows:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="QuarterDim">
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue"
        aggregator="SUM"/>
</mx:OLAPCube>
```

After you call the `OLAPCube.refresh()` method to initialize the cube, it has the following structure:

```
ProductDim                 // Dimension
    Product                    // Hierarchy
        (All)                      // Member
        ColdFusion                 // Member
        Flex                       // Member
        Dreamweaver                // Member
        Illustrator                // Member

    ProductHier                // Hierarchy
        (All)                      // Level
            ColdFusion                 // Member
            Flex                       // Member
            Dreamweaver                // Member
            Illustrator                // Member

        Product                    // Level
            ColdFusion                 // Member
            Flex                       // Member
            Dreamweaver                // Member
            Illustrator                // Member
```

```
QuarterDim                  // Dimension
    Quarter                     // Hierarchy
        (All)                       // Member
        Q1                          // Member
        Q2                          // Member
        Q3                          // Member
        Q4                          // Member

    QuarterHier                 // Hierarchy
        (All)                       // Level
            Q1                          // Member
            Q2                          // Member
            Q3                          // Member
            Q4                          // Member

        Quarter                     // Level
            Q1                          // Member
            Q2                          // Member
            Q3                          // Member
            Q4                          // Member
```

Notice in this cube:

- The ProductDim dimension contains two hierarchies: Product and ProductHier. The cube creates a hierarchy for each level specified by the OLAPLevel class, and for each hierarchy specified by the OLAPHierarchy class. The same is true for the QuarterDim dimension; it also contains two hierarchies.
- The order of the members, meaning the values along each dimension, is based on the order in which the members appear in the flat data. In this example, the quarters appear in order of Q1, Q2, Q3, and Q4 because the flat data was sorted by quarter. However, if the data was not sorted by quarter, the members along the QuarterDim could appear in any order.
- Each dimension contains a single level; therefore the `(All)` level contains the same data as the first level in the hierarchy. For example, the `(All)` level of the ProdcutHier hierarchy contains the same data as the Product level. For a complex cube, the `(All)` level would contain additional information.

For an example of a complex cube, see “Writing a query for a complex OLAP cube” on page 1478.

Defining a query for a simple cube

An OLAP query has the following basic form:

```
// Create an instance of OLAPQuery to represent the query.
var query:OLAPQuery = new OLAPQuery;

// Get the row axis from the query instance.
var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
// Create an OLAPSet instance to configure the axis.
var productSet:OLAPSet = new OLAPSet;
// Use OLAPSet.addElements() or OLAPSet.addElement() to add members to the row axis.
productSet.addElements(...);
// Add the OLAPSet instance to the axis.
rowQueryAxis.addSet(productSet);

// Get the column axis from the query instance, and configure it
// to aggregate the columns by the Quarter dimension.
var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
var quarterSet:OLAPSet= new OLAPSet;
// Use OLAPSet.addElements() or OLAPSet.addElement() to add members to the column axis.
productSet.addElements(...);
colQueryAxis.addSet(quarterSet);
```

You use the following methods and properties to extract information from an OLAP cube based on the attributes of the dimension, and pass it to the `OLAPSet.addElements()` or `OLAPSet.addElement()` method:

Method and property	Description
`OLAPCube.findDimension(name:String):IOLAPDimension`	Returns a dimension of a schema as an instance of the OLAPDimension class.
`OLAPDimension.findAttribute(name:String):IOLAPAttributeHierarchy`	Returns an instance of the IOLAPAttributeHierarchy interface, which is an attribute hierarchy that includes an IList instance of all members of the attribute.
`OLAPDimension.findMember(name:String):IOLAPMember`	Returns an IOLAPMember instance that represents a member with the specified name within the dimension.
`IOLAPAttributeHierarchy.children`	Contains all members of the level as an IList instance, but does not include the (All) member.
`IOLAPAttributeHierarchy.members`	Contains all members of the level as an IList instance, including the (All) member.

The following table shows the information returned for combinations of these methods and properties:

Reference to method and property	Returns
`findDimension("ProductDim").findAttribute("Product").children`	ColdFusion, Flex, Dreamweaver, and Illustrator
`findDimension("ProductDim").findAttribute("Product").members`	(All), ColdFusion, Flex, Dreamweaver, and Illustrator
`findDimension("ProductDim").findMember("Flex")`	Flex
`findDimension("QuarterDim").findAttribute("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findAttribute("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findMember("Q2")`	Q2

Rather than use the `findAttribute()` method, you can drill down through the hierarchy of the cube by calling the following methods:

Method	Description
`OLAPDimension.findHierarchy(name:String):IOLAPHierarchy`	Returns a hierarchy of a dimension as an instance of OLAPHierarchy. You can specify either an OLAPHierarchy or an OLAPLevel instance to this method.
`OLAPHierarchy.findLevel(name:String):IOLAPLevel`	Returns a level of a hierarchy as an instance of OLAPLevel.
`OLAPLevel.findMember(name:String):IList`	Returns an IList instance that contains all IOLAPMember instances that match the String argument. You can then use the `IList.getItemAt()` method to access the any element in the IList.

The following table shows the information returned for different combinations of these methods:

Reference to method	Returns
`findDimension("ProductDim").findhHierarchy("ProductHier").findLevel("Product").children`	ColdFusion, Flex, Dreamweaver, and Illustrator
`findDimension("ProductDim").findHierarchy("ProductHier").findLevel("Product").members`	(All), ColdFusion, Flex, Dreamweaver, and Illustrator
`IOLAPElement(cube.findDimension("ProductDim").findHierarchy("ProductHier").findLevel("Product").findMember("Flex").getItemAt(0))`	Flex. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`IOLAPElement(cube.findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").findMember("Q2").getItemAt(0))`	Q2. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.

In the case of a simple cube, the `OLAPDimension.findHierarchy()`, `OLAPHierarchy.findLevel()`, and `OLAPLevel.findMember()` methods do not provide you with additional functionality from the `OLAPDimension.findAttribute()` and `OLAPDimension.findMember()` methods. They are more commonly used with complex cubes that contain dimensions with multiple levels. For more information, see "Writing a query for a complex OLAP cube" on page 1478.

Add all members to an axis

The most common type of query returns a data aggregation for all members of an attribute of a schema. For example, you define an OLAP cube using a schema that contains a ProductDim and a QuarterDim dimension, as shown in the section "Writing a query for a simple OLAP cube" on page 1471. You then want to generate a query for all products for all quarters. The following query extracts this information:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(
        cube.findDimension("ProductDim").findAttribute("Product").children);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

In this example, the column axis uses the `findDimension()` and `findAttribute()` methods to drill down into the ProductDim and QuarterDim dimensions to extract sales data. For each axis, you use the `IOLAPAttributeHierarchy.children` property to populate it with the all members of the attribute, but do not include the `(All)` member.

This query uses the default measure to populate the query result, where the default measure is the first measure defined in the cube's schema. Therefore, you do have to specify the measure as part of the query. For information on explicitly specifying the measure in the query, see "Creating a query using a nondefault measure" on page 1484.

Notice that in this example, you did not have to specify the hierarchy name, ProductHier and QuarterDim, to extract the member from the schema. It is unnecessary to specify the hierarchy name when you only want to extract data for a single level of a dimension.

However, you could rewrite this example to explicitly drill down through the cube by calling the methods `OLAPDimension.findHierarchy()` and `OLAPHierarchy.findLevel()`, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(cube.findDimension(
        "ProductDim").findHierarchy("ProductHier").findLevel("Product").members);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findHierarchy("QuarterHier").
        findLevel("Quarter").members);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

In this example, you drill down through the dimension to access the Product and Quarter levels. You typically use this technique with complex cubes where you are interested in an explicit member of a dimension.

Add explicit members to a query

In the query shown in the previous section, you obtain sales data for all members of the Product and Quarter levels of the schema. But, what if you only want sales data for a single product, or for a single quarter? When you drill down into a dimension, you can specify the individual member of the dimension that you want to query.

For example, you want to create a query to aggregate quarterly sales data only for Flex, but not for any other products. You therefore use the `OLAPDimension.findMember()` method to specify the name of the member. This method takes a String containing the name of the member, and returns an IOLAPMember instance that defines the member, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElement(cube.findDimension("ProductDim").findMember("Flex"));
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis =
    query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

Notice that in the previous example, you use the `OLAPSet.addElement()` method to add the Flex member to the OLAPSet instance, rather than the `OLAPSet.addElements()` method. This is because you are adding a single member to the axis, rather than multiple members.

The only issue with using the `OLAPDimension.findMember()` method is that it returns the first IOLAPMember instance in the cube that matches the String argument. If you think that you might have multiple instances of a member, you can rewrite this example to explicitly drill down through the cube by calling the `OLAPDimension.findHierarchy()`, `OLAPHierarchy.findLevel()`, and `OLAPLevel.findMember()` methods. This situation is more common with cubes that contain complex dimensions. For more information and examples, see "Writing a query for a complex OLAP cube" on page 1478.

The `OLAPLevel.findMember()` method returns an IList instance that contains all IOLAPMember instances that match the String argument. You can then use the `IList.getItemAt()` method to access any element in the IList, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    // Get the first IOLAPElement instance in the IList instance.
    productSet.addElement(
        IOLAPElement(cube.findDimension("ProductDim").findHierarchy("ProductHier").
        findLevel("Product").findMember("Flex").getItemAt(0)));
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElement(IOLAPMember(cube.findDimension("TimeDim").
        findHierarchy("Month").findMember("January")));
    quarterSet.addElement(IOLAPMember(cube.findDimension("TimeDim").
        findHierarchy("Month").findMember("February")));
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

Writing a query for a complex OLAP cube

In a complex cube, at least one dimension of the schema contains multiple levels. For example, you have flat data that contains three fields:

```
data:Object = {
    product:"ColdFusion"
    year :"2006"
    quarter :"Q1"
    revenue: "100.00",
}
```

The data fields of each record can contain the following values:

- The product field can have the values: ColdFusion, Flex, Dreamweaver, and Illustrator.
- The year field can have the values: 2006 and 2007.

- The quarter field can have the values: Q1, Q2, Q3, and Q4.

In this example, your schema defines two dimensions, with the TimeDim dimension containing levels for year and quarter, as the following code shows:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="TimeDim">
        <mx:OLAPAttribute name="Year" dataField="year"/>
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
            <mx:OLAPLevel attributeName="Year"/>
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue"
        aggregator="SUM"/>
</mx:OLAPCube>
```

The cube has the following structure:

```
ProductDim                  // Dimension
    Product                     // Hierarchy
        (All)                       // Member
        ColdFusion                  // Member
        Flex                        // Member
        Dreamweaver                 // Member
        Illustrator                 // Member

    ProductHier                 // Hierarchy
        (All)                       // Level
            ColdFusion                  // Member
            Flex                        // Member
            Dreamweaver                 // Member
            Illustrator                 // Member

        Product                     // Level
            ColdFusion                  // Member
            Flex                        // Member
            Dreamweaver                 // Member
            Illustrator                 // Member

TimeDim                     // Dimension
    Year                        // Hierarchy
        (All)                       // Member
        2006                        // Member
        2007                            // Member

    Quarter                         // Hierarchy
        (All)                           // Member
        Q1                              // Member
        Q2                              // Member
        Q3                              // Member
        Q4                              // Member

    Time-PeriodHier                 // Hierarchy
        (All)                           // Level
            2006
                Q1
                Q2
                Q3
                Q4
            2007
                Q1
                Q2
                Q3
                Q4

        Year                            // Level
            2006                            // Member
            2007                            // Member

        Quarter                         // Level
            Q1 (having 2006 as a parent) // Member
            Q1 (having 2007 as a parent) // Member
            Q2 (having 2006 as a parent) // Member
            Q2 (having 2007 as a parent) // Member
            Q3 (having 2006 as a parent) // Member
            Q3 (having 2007 as a parent) // Member
            Q4 (having 2006 as a parent) // Member
            Q4 (having 2007 as a parent) // Member
```

Notice in this cube:

- The information for the quarter is added as a hierarchy under the TimeDim dimension, and as a level under the Time-PeriodHier hierarchy.
- The Quarter level under the Time-PeriodHier hierarchy contains multiple entries for each quarter. In this example, there is an entry for each quarter for each year.
- The order of the members, meaning the values along each dimension, is based on the order in which the member values appear in the flat data.

The reason for the multiple entries for a single quarter in the Quarter level under the Time-PeriodHier hierarchy is that a cube has to be able to return results for each quarter for each year. For example, this structure lets you write a query to return data aggregations for all quarters for all years, for all quarters for 2006, or for all quarters for 2007.

Since the ProductDim contains a single level, you can write queries for it in the same way as you did for a simple cube. See “Writing a query for a simple OLAP cube” on page 1471 for more information.

The following table shows the information returned for different ways to access the TimeDim by calling the `findAttribute()` method:

Reference to method and property	**Returns**
`findDimension("TimeDim").findAttribute("Year").children`	2006, 2007
`findDimension("TimeDim").findAttribute("Year").members`	(All), 2006, 2007
`findDimension("TimeDim").findMember("2006")`	2006
`findDimension("TimeDim").findAttribute("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("TimeDim").findAttribute("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`findDimension("TimeDim").findMember("Q2")`	Q2 having 2006 as a parent. This is the first instance of Q2 in the cube.

The following table shows the information returned for different ways to use the `OLAPDimension.findHierarchy()`, `OLAPHierarchy.findLevel()`, and `OLAPLevel.findMember()` methods:

Reference to method and property	**Returns**
`findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Year").children`	2006, 2007
`findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Year").members`	(All), 2006, 2007
`IOLAPElement(cube.findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Year").findMember("2006").getItemAt(0))`	2006
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Quarter").findMember("Q1")`	All Q1 members
`IOLAPElement(cube.findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").findMember("Q2").getItemAt(0))`	Q2 having 2006 as a parent. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.
`IOLAPElement(cube.findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").findMember("Q2").getItemAt(1))`	Q2 having 2007 as a parent. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.

You could add a Month level to the TimeDim, as the following definition of TimeDim shows:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPAttribute name="Month" dataField="month"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
        <mx:OLAPLevel attributeName="Month"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

For this example, the structure of the TimeDim dimension is:

```
TimeDim                     // Dimension
    Year                        // Hierarchy
        (All)                       // Member
        2006                        // Member
        2007                        // Member

    Quarter                     // Hierarchy
        (All)                       // Member
        Q1                          // Member
        Q2                          // Member
        Q3                          // Member
        Q4                          // Member

    Month                       // Hierarchy
        (All)                       // Member
        Jan                         // Member
        Feb                         // Member
        ...                         // Adiitional members for each month

    Time-PeriodHier             // Hierarchy
        (All)                       // Level
            2006
                Q1
                    Jan
                    Feb
                    Mar
                ...
            2007
                Q1
                    Jan
                    Feb
                    Mar
                ...

        Year                        // Level
            2006                        // Member
            2007                        // Member

        Quarter                     // Level
            Q1 (having 2006 as a parent) // Member
            Q1 (having 2007 as a parent) // Member
            Q2 (having 2006 as a parent) // Member
            Q2 (having 2007 as a parent) // Member
            Q3 (having 2006 as a parent) // Member
            Q3 (having 2007 as a parent) // Member
            Q4 (having 2006 as a parent) // Member
            Q4 (having 2007 as a parent) // Member

        Month                       // Level
            Jan (having Q1 having 2006 as a parent) // Member
            Feb (having Q1 having 2006 as a parent) // Member
            Jan (having Q1 having 2007 as a parent) // Member
            Feb (having Q1 having 2007 as a parent) // Member
            ...                                     // Additional members for each possible
                                                    // combination of month, quarter,
                                                    // and year
```

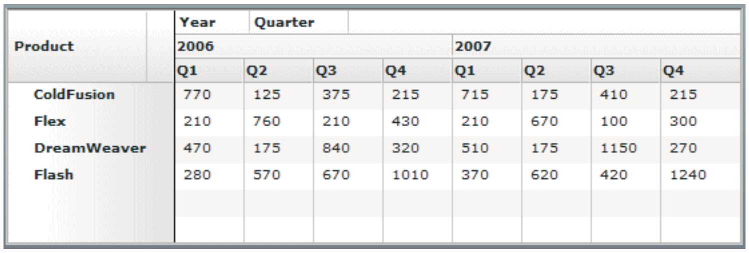
Notice in this cube:

- The Month level contains a value for each month, and for each quarter for each year. Therefore, there should be two entries for January; one for Q1 of 2006 and one for Q1 of 2007.

Creating a multidimensional axis in an OLAPDataGrid control

The OLAPDataGrid control displays information along two axes: row and column. However, limiting yourself to writing queries that return only two dimensions can restrict your ability to examine your data.

To allow you greater flexibility in displaying query results, the OLAPDataGrid control supports hierarchical display along its axis. The following image shows quarterly sales information, grouped by year, displayed in the columns of the OLAPDataGrid control.



	Year	Quarter						
Product	2006				2007			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
ColdFusion	770	125	375	215	715	175	410	215
Flex	210	760	210	430	210	670	100	300
DreamWeaver	470	175	840	320	510	175	1150	270
Flash	280	570	670	1010	370	620	420	1240

To create a multidimensional axis in the OLAPDataGrid control, you create two or more OLAPSet instances for the axis, and then combine the OLAPSet instances by doing a crossjoin or a union, as the following table describes:

Combination type	OLAPSet method	Description
Crossjoin	`crossJoin()`	Creates a crossjoin of two OLAPSet instances, where a crossjoin contains all possible combinations of the two sets. A crossjoin is also called a cross product of the members of the two different sets.
Union	`union()`	Creates a union of two OLAPSet instances.

In the following example, you create an OLAPSet instance for the year, and then crossjoin that set with the OLAPSet instance for quarter to create the OLAPDataGrid control shown in the previous image:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    var query:OLAPQuery = new OLAPQuery;

    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(
        cube.findDimension("ProductDim").findAttribute("Product").children);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var yearSet:OLAPSet= new OLAPSet;
    yearSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Year").children);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(yearSet.crossJoin(quarterSet));

    return query;
}
```

Creating a slicer axis

An OLAP query can have three axes: row, column, and slicer. A slicer axis lets you reduce the size of the query results, often to reduce the dimensionality of the results from a dimension greater than two so that you can display the results. Another common use of a slicer axis is to aggregate data on a measure other than the default measure.

Creating a query using a nondefault measure

The following schema defines two measures for the points in the cube: Revenue and Cost. The first measure defined in the schema is the default measure, and it is the data that is aggregated by a query if you do not explicitly specify the measure.

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="QuarterDim">
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue" aggregator="MAX"/>
    <mx:OLAPMeasure name="Cost"
        dataField="cost" aggregator="MIN"/>
</mx:OLAPCube>
```

To aggregate by the Cost measure, you create a slicer axis to explicitly specify the measure for the query, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(
        cube.findDimension("ProductDim").findAttribute("Product").children);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Month").children);
    colQueryAxis.addSet(quarterSet);

    // Create the slicer axis.
    var slicerQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.SLICER_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var costSet:OLAPSet= new OLAPSet;
    // Use OLAPDimension.findMember() to add the Cost measure.
    costSet.addElement(cube.findDimension("Measures").findMember("Cost"));
    slicerQueryAxis.addSet(costSet);

    return query;
}
```

In this example, you use the keyword `Measures` to identify the measure dimension, and then use the `OLAPDimension.findMember()` method to access the Cost measure.

Using a slicer axis to reduce the dimensionality of the query result

The OLAPDataGrid control displays information in two dimensions along its row and column axis. However, to display product sales by region and by month, you require a three-dimensional display: one dimension each for product, region, and month.

For example, your data has the following format:

```
data:Object = {
    customer:"IBM",
    country:"US",
    state:"MA",
    region:"NewEngland",
    product:"ColdFusion",
    year:2005,
    quarter:"Q1"
    month:"January",
    revenue: 12,575.00,
    cost: 500
}
```

The examples use the following OLAP schema to represent this data in an OLAP cube:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="CustomerDim">
        <mx:OLAPAttribute name="Customer" dataField="customer"/>
        <mx:OLAPHierarchy name="CustomerHier"
            hasAll="true">
            <mx:OLAPLevel attributeName="Customer"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier"
            hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>
    <mx:OLAPDimension name="TimeDim">
        <mx:OLAPAttribute name="Year" dataField="year"/>
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPAttribute name="Month" dataField="month"/>
        <mx:OLAPHierarchy name="Time-Period"
            hasAll="true">
            <mx:OLAPLevel attributeName="Year"/>
            <mx:OLAPLevel attributeName="Quarter"/>
            <mx:OLAPLevel attributeName="Month"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>
    <mx:OLAPDimension name="GeographyDim">
        <mx:OLAPAttribute name="Country" dataField="country"/>
        <mx:OLAPAttribute name="Region" dataField="region"/>
        <mx:OLAPAttribute name="State" dataField="state"/>
        <mx:OLAPHierarchy name="Country-Region-State"
            hasAll="true">
            <mx:OLAPLevel attributeName="Country"/>
            <mx:OLAPLevel attributeName="Region"/>
            <mx:OLAPLevel attributeName="State"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>
    <mx:OLAPMeasure name="Revenue"
        dataField="revenue"
        aggregator="SUM"/>
    <mx:OLAPMeasure name="Cost"
     dataField="cost"
     aggregator="SUM"/>
</mx:OLAPCube>
```

You can use a slicer axis to filter the results of a query for display in two dimensions. In this example, you set the row axis to the region and the column axis to the month. You then define a slicer axis to specify the product as Flex so that you end up with a two-dimensional table of sales by region and month for Flex.

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(
        cube.findDimension("GeographyDim").findAttribute("Region").children);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Month").children);
    colQueryAxis.addSet(quarterSet);

    // Create the slicer axis.
    var slicerQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.SLICER_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var flexSet:OLAPSet= new OLAPSet;
    flexSet.addElement(
        IOLAPElement(cube.findDimension("ProductDim").findHierarchy("ProductHier").
        findLevel("Product").findMember("Flex").getItemAt(0)));
    slicerQueryAxis.addSet(flexSet);

    return query;
}
```

You can specify more than one member for the slicer access. For example, if your cube contains information on different product such as Flex and Adobe® Flash®, you could create the two slicer axes as the following example shows:

```
// Create the slicer axis.
var slicerQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.SLICER_AXIS);

// Create an OLAPSet instance to configure the axis.
var sliceSet:OLAPSet= new OLAPSet;
sliceSet.addElement(IOLAPElement(cube.findDimension("ProductDim").
    findHierarchy("ProductHier").findLevel("Product").findMember("Flex").getItemAt(0)));

sliceSet.addElement(IOLAPElement(cube.findDimension("ProductDim").
    findHierarchy("ProductHier").findLevel("Product").findMember("Flash").getItemAt(0)));

slicerQueryAxis.addSet(sliceSet);
```

In this example, your query result would show sales of Flex and Flash for each region and month.

Configuring the display of an OLAPDataGrid

To control the display of an OLAPDataGrid control, you can apply styles to the cells of the control by using a callback function, or use item renderers to control the display.

Styling cells of the OLAPDataGrid control by using a callback function

To control the styling of a cell by using a callback function, use the `OLAPDataGrid.styleFunction` property to specify the callback function. The callback function must have the following signature:

```
function_name(row:IOLAPAxisPosition, column:IOLAPAxisPosition, value:Number):Object
```

where `row` is the IOLAPAxisPosition associated with this cell on the row axis, `column` is the IOLAPAxisPosition associated with this cell on the column axis, and `value` is the cell value.

The function returns an Object that contains one or more *styleName:value* pairs to specify a style setting, or null. The *styleName* field contains the name of a style property, such as `color`, and the *value* field contains the value for the style property, such as 0x00FF00. For example, you could return two styles using the following syntax:

```
{color:0xFF0000, fontWeight:"bold"}
```

The OLAPDataGrid control invokes the callback function when it updates its display, such as when the control is first drawn on application start up, or when you call the `invalidateList()` method.

The following example modifies the example shown in the section "Example using the OLAPDataGrid control" on page 1460 to add a callback function to display all cells with a value greater than 1000 in green:

```
<fx:Script>
    <![CDATA[
        ...

        // Callback function that hightlights in green
        // all cells with a value greater than or equal to 1000.
        public function myStyleFunc(row:IOLAPAxisPosition, column:IOLAPAxisPosition,
            value:Number):Object
        {
            if (value >= 1000)
            return {color:0x00FF00};

            // Return null if value is less than 1000.
            return null;
        }
        ]]>
</fx:Script>

...

<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%"
    styleFunction="myStyleFunc"/>
```

Using an item renderer with the OLAPDataGrid control

You customize the appearance and behavior of cells in an OLAPDataGrid control by creating custom item renderers.

For an introduction to item renderers and item editors, see "MX item renderers and item editors" on page 1006.

To use an item renderer with the OLAPDataGrid control, you assign the item renderer to the OLAPDataGrid control itself, not to a specific column, by using the `OLAPDataGrid.itemRendererProviders` property. The `itemRendererProviders` property contains an Array of OLAPDataGridItemRendererProvider instances, where each OLAPDataGridItemRendererProvider instance defines the characteristics for a single item renderer.

You can use the `OLAPDataGridRendererProvider.renderer` property to assign an item renderer to the OLAPDataGrid control, much in the way that you can by using the `AdvancedDataGrid.rendererProviders` property, or can use the `OLAPDataGridRendererProvider.formatter` property to assign a formatter class. The following example modifies the example shown in the section "Example using the OLAPDataGrid control" on page 1460 to add an item renderer that applies the CurrencyFormatter formatter to each cell in the control:

```
...
<mx:CurrencyFormatter id="usdFormatter" precision="2"
    currencySymbol="$" decimalSeparatorFrom="."
    decimalSeparatorTo="." useNegativeSign="true"
    useThousandsSeparator="true" alignSymbol="left"/>

...

<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%">

    <mx:itemRendererProviders>
        <mx:OLAPDataGridItemRendererProvider
            uniqueName="[QuarterDim].[Quarter]"
            type="{OLAPDataGrid.OLAP_HIERARCHY}"
            formatter="{usdFormatter}"/>
    </mx:itemRendererProviders>
</mx:OLAPDataGrid>
```

To assign the item renderer, use the `uniqueName` and `type` properties of the OLAPDataGridItemRendererProvider class. The `uniqueName` property specifies the elements of the query, and therefore the corresponding cells of the OLAPDataGrid control, that you modify by using the item renderer. The `type` properly specifies the element type, such as dimension, hierarchy, or level of the element specified by the `uniqueName` property.

When assigning a formatter to the OLAPDataGridItemRendererProvider, you must specify a subclass of the mx.formatters.Formatter class; for example, the MX formatters such as CurrencyFormatter or NumberFormatter. The formatter cannot be a Spark formatter.

The function that defines the query for this example is shown below:

```
// Create the OLAP query.
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    // Add the Product to the row to aggregate data
    // by the Product dimension.
    productSet.addElements(
        cube.findDimension("ProductDim").findAttribute("Product").children);
    // Add the OLAPSet instance to the axis.
    rowQueryAxis.addSet(productSet);

    // Get the column axis from the query instance, and configure it
    // to aggregate the columns by the Quarter dimension.
    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

Notice that the axis for the QuarterDim creates a query for the Quarter hierarchy of the QuarterDim dimension. Therefore, you set the `uniqueName` property to `[QuarterDim].[Quarter]`, and set the type property to `OLAPDataGrid.OLAP_HIERARCHY`. For more information on the structure of the `OLAP` code for this example, see "Writing a query for a simple OLAP cube" on page 1471.

You can modify this example to drill down through the cube by specifying the query as shown below for the QuarterDim:

```
quarterSet.addElements(
    cube.findDimension("QuarterDim").findHierarchy("QuarterHier").
    findLevel("Quarter").members);
```

You therefore modify the `uniqueName` and `type` properties as shown below:

```
<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%">

    <mx:itemRendererProviders>
        <mx:OLAPDataGridItemRendererProvider
            uniqueName="[QuarterDim].[QuarterHier].[Quarter]"
            type="{OLAPDataGrid.OLAP_LEVEL}"
            formatter="{usdFormatter}"/>
    </mx:itemRendererProviders>
</mx:OLAPDataGrid>
```

Notice in this example that the query for the QuarterDim specifies the dimension, hierarchy, and level of the OLAP cube. Therefore, you modify the `uniqueName` property to match this structure, and set the `type` property to `OLAPDataGrid.OLAP_LEVEL`.

Resolving item renderer conflicts

Each cell in an OLAPDataGrid control is a result of an intersection between the members along a row and the members along a column of the control. However, when you assign an item renderer to an OLAPDataGrid control, you only specify the `uniqueName` and `type` properties for one of the dimensions, either row or column. Therefore, you can create a situation where two different item renderers are assigned to the same cell of the control.

In case of a conflict between two or more item renderers, the OLAPDataGrid control applies the item renderer based on the following priorities:

1. `type` = `OLAPDataGrid.OLAP_MEMBER`
2. `type` = `OLAPDataGrid.OLAP_LEVEL`
3. `type` = `OLAPDataGrid.OLAP_HIERARCHY`
4. `type` = `OLAPDataGrid.OLAP_DIMENSION`

Therefore, if an item renderer with a `type` value of `OLAPDataGrid.OLAP_LEVEL` and an item renderer with a `type` value of `OLAPDataGrid.OLAP_HIERARCHY` are applied to the same cell, the OLAPDataGrid control applies the item renderer with a `type` value of `OLAPDataGrid.OLAP_LEVEL`.

If two item renderers have the same value for the `type` property, the OLAPDataGrid control determines which renderer more closely matches the item, and uses it.

Using an item renderer to apply styles to an OLAPDataGrid control

You can use an item renderer to apply styles to specific cells in the OLAPDataGrid control, as the following example shows:

```
<fx:Style>
    .cellStyle
    {
        color:#ff0000;
        fontWeight:"bold"
    }
</fx:Style>

    ...

<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%">

    <mx:itemRendererProviders>
        <mx:OLAPDataGridItemRendererProvider
            uniqueName="[QuarterDim].[Quarter]"
            type="{OLAPDataGrid.OLAP_HIERARCHY}"
            formatter="{usdFormatter}"
            styleName="cellStyle"/>
    </mx:itemRendererProviders>
</mx:OLAPDataGrid>
```

In this example, you create a style definition, then apply it to the cells of the OLAPDataGrid control so that it applies to the same cells as the CurrencyFormatter formatter.

Chapter 6: Enhancing the user interface

Styles and themes

Styles are useful for defining the look and feel of your applications built with Adobe® Flex®. You can use them to change the appearance of a single component, or apply them across all components.

Flex 4 deemphasizes the use of style properties for some tasks. Instead of styling individual components, or classes of components, you can use custom skins to perform most operations that change the look and feel of your applications. This includes adding background colors and images, borders, and other properties that were styleable in Flex 3 applications. For more information on skinning, see "Spark Skinning" on page 1602.

Peter deHaan's blog, FlexExamples.com has many examples of using styles and themes in Flex.

About styles

You modify the appearance of components through style properties. In Flex, some styles are inherited by children from their parent containers, and across style types and classes. This means that you can define a style once, and then apply that style to all controls of a single type or to a set of controls. In addition, you can override individual properties for each control at a local, component, or global level, giving you great flexibility in controlling the appearance of your applications.

The style properties that you can set on a component depend on the component's architecture. For example, the components in the MX component set (mx.controls.* and mx.containers.* packages) take one set of styles. Components in the Spark component set (spark.components.* package) allow an entirely different set of styles.

This section introduces you to applying styles to controls. It also provides a primer for using Cascading Style Sheets (CSS), an overview of the style value formats (`Length`, `Color`, and `Time`), and describes style inheritance. Subsequent sections provide detailed information about different ways of applying styles in Flex.

Flex does not support controlling all aspects of component layout with CSS. Properties such as `x`, `y`, `width,` and `height` are properties, not styles, of the UIComponent class, and therefore cannot be set in CSS.

Using styles in Flex

There are many ways to apply styles in Flex. Some provide more granular control and can be approached programmatically. Others are not as flexible, but can require less computation and therefor be better for your application's performance.

For MX components, to determine if an object's properties can be styled by using CSS, check to see if the class or its parent implements the IStyleClient interface. If it does, you can set the values of the object's style properties with CSS. If it does not, the class does not participate in the styles subsystem and you therefore cannot set its properties with CSS. In that case, the properties are public properties of the class and not style properties.

For Spark components, many properties that were formerly available as styles are now properties that you set on the Spark component's skin. As a result, to do something like create a border around a container, you edit the container's skin class rather than add a style property. For more information about skinning Spark components, see "Spark Skinning" on page 1602.

When applying styles, you must be aware of which properties your theme supports. The default theme in Flex does not support all style properties. To determine if a style property is supported by the theme you are using, view the style's entry in ActionScript 3.0 Reference for the Adobe Flash Platform. If the style property is limited to a theme, the Theme property will appear in the style's description, with the name of the supported themes next to it. For more information, see "About supported styles" on page 1532.

External style sheets

Use CSS to apply styles to a document or across entire applications. You can point to a style sheet without invoking ActionScript. This is the most concise method of applying styles, but can also be the least flexible. Style sheets can define global styles that are inherited by all controls, or individual classes of styles that only certain controls use.

The following example applies the external style sheet myStyle.css to the current document:

```
<fx:Style source="myStyle.css"/>
```

For more information, see "Using external style sheets" on page 1533.

Flex includes global style sheets inside the spark.swc and framework.swc files. These files contain style definitions for the global class selector, and type selectors for most components. For more information about the defaults.css files, see "About the default style sheets" on page 1534.

Flex also includes several other style sheets that each provide a unique look and feel. For more information, see "About the included theme files" on page 1565.

Local style definitions

Use the `<fx:Style>` tag to define styles that apply to the current document and its children. You define styles in the `<fx:Style>` tag using CSS syntax and can define styles that apply to all instances of a control or to individual controls. The following example defines a new style and applies it to only the myButton control:

```
<?xml version="1.0"?>
<!-- styles/ClassSelectorAgain.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        .myFontStyle {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>
    <s:VGroup>
        <!-- This button has the custom style applied to it. -->
        <s:Button id="myButton" styleName="myFontStyle" label="Click Me"/>
        <!-- This button uses default styles. -->
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

The following example defines a new style that applies to all instances of the Button class in the Spark namespace:

```
<?xml version="1.0"?>
<!-- styles/TypeSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>
    <s:VGroup>
        <s:Button id="myButton" label="Click Me"/>
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

For more information, see “Using local style definitions” on page 1535.

StyleManager class

Use the StyleManager class to apply styles to all classes or all instances of specified classes. You can access the top-level StyleManager by using the `styleManager` property of the Application object.

The following example sets the `fontSize` style to 15 and the `color` to `0x9933FF` on all Button controls in the Spark namespace:

```
<?xml version="1.0"?>
<!-- styles/StyleManagerExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script><![CDATA[
        public function initApp():void {

styleManager.getStyleDeclaration("spark.components.Button").setStyle("fontSize",15);

styleManager.getStyleDeclaration("spark.components.Button").setStyle("color",0x9933FF);
        }
    ]]></fx:Script>
    <s:VGroup>
        <s:Button id="myButton" label="Click Me"/>
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

You can also use the CSSStyleDeclaration object to build run-time style sheets, and then apply them with the StyleManager’s `setStyleDeclaration()` method.

For more information, see “Using the StyleManager class” on page 1537.

getStyle() and setStyle() methods

Use the setStyle() and getStyle() methods to manipulate style properties on instances of controls. Using these methods to apply styles requires a greater amount of processing power on the client than using style sheets but provides more granular control over how styles are applied.

The following example sets the `fontSize` to 15 and the `color` to `0x9933FF` on only the myButton instance:

```
<?xml version="1.0"?>
<!-- styles/SetStyleExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script><![CDATA[
        public function initApp():void {
            myButton.setStyle("fontSize",15);
            myButton.setStyle("color",0x9933FF);
        }
    ]]></fx:Script>
    <s:VGroup>
        <s:Button id="myButton" label="Click Me"/>
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

For more information, see “Using the setStyle() and getStyle() methods” on page 1540.

Inline styles

Use attributes of MXML tags to apply style properties. These properties apply only to the instance of the control. This is the most efficient method of applying instance properties because no ActionScript code blocks or method calls are required.

The following example sets the `fontSize` to `15` and the `color` to `0x9933FF` on the myButton instance:

```
<?xml version="1.0"?>
<!-- styles/InlineExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:VGroup>
        <!-- This button uses custom inline styles. -->
        <s:Button id="myButton" color="0x9933FF" fontSize="15" label="Click Me"/>
        <!-- This button uses default styles. -->
        <s:Button id="myOtherButton" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

In an MXML tag, you must use the camel-case version of the style property. For example, you must use “`fontSize`” rather than “`font-size`” (the CSS convention) in the previous example. For more information on style property names, see “About selector names” on page 1506.

As with other style properties, you can bind inline style properties to variables.

For more information, see “Using inline styles” on page 1546.

Setting global styles

To set a global style that is inheritable, you set its value on the container. All children of that container, including other containers, inherit the value of that style property. For example, if you set the `color` of a Panel container to green, the labels for all buttons in the Panel container are also green, unless those buttons override that color.

To use CSS to apply a noninheritable style globally, you can use the `global` selector. For more information, see "Using the global selector" on page 1536.

When using Spark components, the easiest way to set global styles is to set the value of the style property on the Application class, as long as that style is inheritable. To do this, you use the `FlexGlobals.topLevelApplication.application` property to reference the class, and then call the `setStyle()` method on it. For more information, see "Using themes" on page 1562.

If you apply a noninheritable style to a parent container, only the container uses that style. Children of that container do not use the values of noninheritable styles.

By using global styles, you can apply noninheritable styles to all controls that do not explicitly override that style. Flex provides the following ways to apply styles globally:

- StyleManager `global` style
- CSS `global` selector

The StyleManager lets you apply styles to all controls using the `global` style. You can access the top-level StyleManager by using the `styleManager` property of the Application object. For more information, see "Using the StyleManager class" on page 1537.

You can also apply global styles using the `global` selector in your CSS style definitions. These are located either in external CSS style sheets or in an `<fx:Style>` tag. For more information, see "Using the global selector" on page 1536.

About style value formats

Style properties can be of types String or Number. They can also be Arrays of these types. In addition to a type, style properties also have a format (`Length`, `Time`, or `Color`) that describes the valid values of the property. Some properties appear to be of type Boolean (taking a value of `true` or `false`), but these are Strings that are interpreted as Boolean values. This section describes these formats.

Length format

The `Length` format applies to any style property that takes a size value, such as the size of a font (or `fontSize`). `Length` is of type Number.

The `Length` type has the following syntax:

```
length[length_unit]
```

The following example defines the `fontSize` property with a value of 20 pixels:

```
<?xml version="1.0"?>
<!-- styles/LengthFormat.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
     .myFontStyle {
        fontSize: 20px;
        color: #9933FF;
     }
  </fx:Style>
  <s:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
</s:Application>
```

If you do not specify a unit, the unit is assumed to be pixels. The following example defines two styles with the same font size:

```
<?xml version="1.0"?>
<!-- styles/LengthFormat2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
     .myFontStyle {
        fontSize: 20px;
        color: #9933FF;
     }
     .myOtherFontStyle {
        fontSize: 20;
        color: #9933FF;
     }
  </fx:Style>
  <s:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
  <s:Button id="myButton2" styleName="myOtherFontStyle" label="Click Here"/>
</s:Application>
```

***Note:** Spaces are not allowed between the length value and the unit.*

The following table describes the supported `length` units:

Unit	Scale	Description
px	Relative	Pixels.
in	Absolute	Inches.
cm	Absolute	Centimeters.
mm	Absolute	Millimeters.
pt	Absolute	Points.
pc	Absolute	Picas.

***Note:** The numeric values specified for font size in Flex are actual character sizes in the chosen units. These values are not equivalent to the relative font size specified in HTML using the `<font>` tag.*

Flex does not support the em and ex units. You can convert these to px units by using the following scales:

1em = 10.06667px

1ex = 6px

In Flex, all lengths are converted to pixels prior to being displayed. In this conversion, Flex assumes that an inch equals 72 pixels. All other lengths are based on that assumption. For example, 1 cm is equal to 1/2.54 of an inch. To get the number of pixels in 1 cm, multiply 1 by 72, and divide by 2.54.

When you use inline styles, Flex ignores units and uses pixels as the default.

The `fontSize` style property allows a set of keywords in addition to numbered units. You can use the following keywords when setting the `fontSize` style property. The exact sizes are defined by the client browser:

- `xx-small`
- `x-small`
- `small`
- `medium`
- `large`
- `x-large`
- `xx-large`

The following example class selector defines the `fontSize` as `x-small`:

```
<?xml version="1.0"?>
<!-- styles/SmallFont.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Style>
     .smallFont {
        fontFamily: Arial, Helvetica, "_sans";
        fontSize: x-small;
     }
  </fx:Style>
  <s:Button id="myButton" styleName="smallFont" label="Click Here"/>
</s:Application>
```

Time format

You use the `Time` format for component properties that move or have built-in effects, such as the ComboBox component when it drops down and pops up. The `Time` format is of type Number and is represented in milliseconds. Do not specify the units when entering a value in the `Time` format.

The following example sets the `openDuration` style property of the myTree control to 1000 milliseconds. The default value is 0, so this example opens the tree nodes considerably more slowly than normal.

```
<?xml version="1.0"?>
<!-- styles/TimeFormat.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script><![CDATA[
        public function initApp():void {
            myTree.setStyle("openDuration", 1000);
            myTree.setStyle("paddingLeft", 50);
        }
    ]]></fx:Script>
    <fx:Declarations>
        <fx:XMLList id="treeData">
            <node label="Mail Box">
                <node label="Inbox">
                    <node label="Marketing"/>
                    <node label="Product Management"/>
                    <node label="Personal"/>
                </node>
                <node label="Outbox">
                    <node label="Professional"/>
                    <node label="Personal"/>
                </node>
                <node label="Spam"/>
                <node label="Sent"/>
            </node>
        </fx:XMLList>
    </fx:Declarations>
    <mx:Panel title="Tree Control Example" width="100%">
        <mx:Tree id="myTree" width="100%" labelField="@label" dataProvider="{treeData}"/>
    </mx:Panel>
</s:Application>
```

Color format

You define `Color` in several formats. You can use most of the formats only in the CSS style definitions. The following table describes the recognized `Color` formats for a style property:

Format	**Description**
hexadecimal	Hexadecimal colors are represented by a six-digit code preceded by either a zero and small x (`0x`) or a pound sign (#). The range of valid values is `0x000000` to `0xFFFFFF` (or `#000000` to `#FFFFFF`). You use the `0x` prefix when defining colors in calls to the `setStyle()` method and in MXML tags. You use the # prefix in CSS style sheets and in `<fx:Style>` tag blocks.

Format	**Description**
RGB	RGB colors are a mixture of the colors red, green, and blue, and are represented in percentages of the color's saturation. The format for setting RGB colors is `color:rgb(x%, y%, z%)`, where the first value is the percentage of red saturation, the second value is the percentage of green saturation, and the third value is the percentage of blue saturation. You can use the RGB format only in style sheet definitions.
8-bit octet RGB	The 8-bit octet RGB colors are red, green, and blue values from 1 to 255. The format of 8-bit octet colors is `[0-255],[0-255],[0-255]`. You can use the 8-bit octet RGB format only in style sheet definitions.
VGA color names	VGA color names are a set of 16 basic colors supported by all browsers that support CSS. The available color names are `Aqua`, `Black`, `Blue`, `Fuchsia`, `Gray`, `Green`, `Lime`, `Maroon`, `Navy`, `Olive`, `Purple`, `Red`, `Silver`, `Teal`, `White`, `Yellow`. You can use the VGA color names format in style sheet definitions and inline style declarations. VGA color names are not case-sensitive.

Color formats are of type Number. When you specify a format such as a VGA color name, Flex converts that String to a Number.

CSS style definitions and the `<fx:Style>` tag support the following color value formats:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatCSS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Style>
     .myStyle {
        themeColor: #6666CC;    /* CSS hexadecimal format */
        color: Blue;        /* VGA color name */
     }
  </fx:Style>

  <s:Button id="myButton" styleName="myStyle" label="Click Here"/>

</s:Application>
```

You can use the following color value formats when setting the styles inline or using the `setStyle()` method:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatStyleManager.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
  <fx:Script><![CDATA[
     public function initApp():void {
        styleManager.getStyleDeclaration("spark.components.Button").setStyle("color","Blue");
     }
     public function changeStyles(e:Event):void {
        // Check against "255" here, because that is the numeric value of "Blue".
        if (e.currentTarget.getStyle("color") == 255) {
            e.currentTarget.setStyle("color", "Red");
        } else {
            e.currentTarget.setStyle("color", "Blue");
        }
     }
  ]]></fx:Script>
  <s:Button id="myButton" label="Click Here"
        click="changeStyles(event)"/>
</s:Application>
```

Using Arrays for style properties

Some controls accept arrays of colors. For example, the Tree control's `depthColors` style property can use a different background color for each level in the tree. To assign colors to a property in an Array, add the items in a comma-separated list to the property's definition. The index is assigned to each entry in the order that it appears in the list.

The following example defines Arrays of colors for properties of the `Tree` type selector:

```
<?xml version="1.0"?>
<!-- styles/ArraysOfColors.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";
     mx|Tree {
        depthColors: #FFCC33, #FFCC99, #CC9900;
        alternatingItemColors: red, green;
     }
  </fx:Style>
    <fx:Declarations>
      <fx:XMLList id="treeData">
         <node label="Mail Box">
            <node label="Inbox">
               <node label="Marketing"/>
               <node label="Product Management"/>
               <node label="Personal"/>
            </node>
            <node label="Outbox">
               <node label="Professional"/>
               <node label="Personal"/>
            </node>
            <node label="Spam"/>
            <node label="Sent"/>
         </node>
      </fx:XMLList>
    </fx:Declarations>
  <s:Panel title="Tree Control Example" width="100%">
        <mx:Tree id="myTree" width="100%" labelField="@label" dataProvider="{treeData}"/>
  </s:Panel>
</s:Application>
```

In this example, only `depthColors` will be seen. The `alternatingItemColors` property is visible only if `depthColors` is not set. Both are presented here for illustrative purposes only.

You can define the Array in ActionScript by using a comma-separated list of values surrounded by braces, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/SetStyleArray.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()"
    height="500">

  <fx:Script>
    <![CDATA[
     public function initApp():void {
        myTree.setStyle("depthColors",[0xFFCC33, 0xFFCC99, 0xCC9900]);
        myTree.setStyle("alternatingItemColors",["red", "green"]);
     }
    ]]>
  </fx:Script>
  <fx:Declarations>
      <fx:XMLList id="treeData">
         <node label="Mail Box">
            <node label="Inbox">
               <node label="Marketing"/>
               <node label="Product Management"/>
               <node label="Personal"/>
            </node>
            <node label="Outbox">
               <node label="Professional"/>
               <node label="Personal"/>
            </node>
            <node label="Spam"/>
            <node label="Sent"/>
         </node>
      </fx:XMLList>
  </fx:Declarations>
  <s:Panel title="Tree Control Example" width="100%">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:Tree id="myTree"
           width="100%"
           labelField="@label"
           dataProvider="{treeData}"/>
        <mx:Tree id="myOtherTree"
           width="100%"
           labelField="@label"
           dataProvider="{treeData}"
           depthColors="[0xFFCC33, 0xFFCC99, 0xCC9900]"
           alternatingItemColors="['red', 'green']"/>
  </s:Panel>
</s:Application>
```

This example also shows that you can set the properties that use Arrays inline.

Finally, you can set the values of the Array in MXML syntax and apply those values inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ArrayOfColorsMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
      <fx:Array id="myDepthColors">
         <fx:Object>0xFFCC33</fx:Object>
         <fx:Object>0xFFCC99</fx:Object>
         <fx:Object>0xCC9900</fx:Object>
      </fx:Array>
      <fx:Array id="myAlternatingRowColors">
         <fx:Object>red</fx:Object>
         <fx:Object>green</fx:Object>
      </fx:Array>
      <fx:XMLList id="treeData">
         <node label="Mail Box">
            <node label="Inbox">
               <node label="Marketing"/>
               <node label="Product Management"/>
               <node label="Personal"/>
            </node>
            <node label="Outbox">
               <node label="Professional"/>
               <node label="Personal"/>
            </node>
            <node label="Spam"/>
            <node label="Sent"/>
         </node>
      </fx:XMLList>
    </fx:Declarations>
  <s:Panel title="Tree Control Example" width="50%">
        <mx:Tree id="myTree"
           width="100%"
           labelField="@label"
           dataProvider="{treeData}"
           depthColors="{myDepthColors}"
           alternatingItemColors="{myAlternatingRowColors}"/>
  </s:Panel>
</s:Application>
```

Using Cascading Style Sheets

Cascading Style Sheets (CSS) are a standard mechanism for declaring text styles in HTML and most scripting languages. A style sheet is a collection of formatting rules for types of components or classes that include sets of components. Flex supports the use of CSS syntax and styles to apply styles to components.

The dremsus.com blog has several examples of using CSS with Flex.

In CSS syntax, each declaration associates a style name, or *selector*, with one or more style properties and their values. You define multiple style properties in each selector by separating each property with a semicolon. For example, the following style defines a selector named `myFontStyle`:

```
<?xml version="1.0"?>
<!-- styles/ClassSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        .myFontStyle {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>
    <s:VGroup>
        <!-- This button has the custom style applied to it. -->
        <s:Button id="myButton" styleName="myFontStyle" label="Click Me"/>
        <!-- This button uses default styles. -->
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

In this example, `myFontStyle` defines a new class of styles, so it is called a *class selector*. In the markup, you can explicitly apply the `myFontStyle` style to a control or class of controls.

A *type selector* implicitly applies itself to all components of a particular type, as well as all subclasses of that type. If you use type selectors, then you must be sure to define namespaces with the `@namespace` directive in the CSS so that Flex. This is because classes in different packages often share the same class name.

The following example defines a type selector named `Button`:

```
<?xml version="1.0"?>
<!-- styles/TypeSelectorAgain.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>
    <s:VGroup>
        <s:Button id="myButton" label="Click Me"/>
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

Flex applies this style to all Button controls, and all subclasses of Button controls in the Spark namespace. If you define a type selector on a container, that style applies to all children of that container if the style is an inheriting style.

Flex also supports id, descendent, and pseudo selectors.

When discussing CSS in Flex, the *subject* of a selector is the right-most simple type selector in a potentially-complex selector expression. In the following example, the Button is the subject of the selectors:

```
VBox Panel Button#button12 {
    color: #DDDDDD;
}
VBox.special Button {
    color: #CCCCCC;
}
Button.special {
    color: #BBBBBB;
}
```

When determining styles for a new component instance, Flex examines all the parent classes looking for type selectors. Flex applies settings in all type selectors, not just the exact match. For example, suppose that class MyButton extends Button. For an instance of MyButton, Flex first checks for a MyButton type selector. Flex applies styles in the MyButton type selector, and then checks for a Button type selector. Flex applies styles in the Button selector, and then checks for a UIComponent type selector. Flex stops at UIComponent. Flex does not continue up the parent chain past UIComponent because Flex does not support type selectors for Sprite (the parent of UIComponent) or any of Sprite's parent classes, up to the base Object class.

***Note:** The names of class selectors cannot include hyphens in Flex. If you use a hyphenated class selector name, such as my-class-selector, Flex ignores the style.*

You can programmatically define new class and type selectors using the StyleManager class. You can access the top-level StyleManager by using the `styleManager` property of the Application object. For more information, see "Using the StyleManager class" on page 1537.

About selector names

When applying style properties with CSS in a `<fx:Style>` block or in an external style sheet, the best practice is to use camel-case without hyphens for the style property, as in `fontWeight` and `fontFamily` (rather than `font-weight` and `font-family`). This matches the convention of using camel-case property names in MXML.

To make development easier, however, Flex supports both the camel-case and hyphenated syntax in style sheets, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/CamelCase.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

    <fx:Style>
        .myFontStyle {
            fontSize: 15; /* Note the camelCase. */
        }
        .myOtherFontStyle {
            font-size: 15; /* Note the hyphen. */
        }
    </fx:Style>
    <s:Button id="myButton" styleName="myFontStyle" label="Click Me"/>
    <s:Button id="myButton2" styleName="myOtherFontStyle" label="Click Me"/>
</s:Application>
```

In ActionScript or an MXML tag, you cannot use hyphenated style property names, so you must use the camel-case version of the style property. For the style name in a style sheet, you cannot use a hyphenated name, as the following example shows:

```
.myClass { ... } /* Valid style name */
.my-class { ... } /* Not a valid style name */
```

About namespaces in CSS

Some Spark and MX components share the same local name. For example, there is a Spark Button component (in the spark.components.* package) and an MX Button component (in the mx.controls.* package). To distinguish between different components that share the same name, you specify namespaces in your CSS that apply to types.

For example, you can specify that a particular selector apply to all components in the Spark namespace only. To specify a namespace in CSS, you declare the namespace with the @namespace directive, followed by the namespace's library in quotation marks. The following example defines the Spark namespace and uses the "s" as an identifier:

```
@namespace s "library://ns.adobe.com/flex/spark";
```

The following are valid namespace directives in Flex 4 that refer to the manifests in the related SWC files:

- `library://ns.adobe.com/flex/spark`
- `library://ns.adobe.com/flex/mx`

If you do not use type selectors in your style sheets, then you are not required to use the @namespace rule.

After you specify a namespace's identifier, you can use it in CSS selectors. The following example uses the Spark namespace for the Button components and the MX namespace for the Box containers:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/NamespaceIdentifierExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComps="*"
    >
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
          @namespace s "library://ns.adobe.com/flex/spark";
          @namespace mx "library://ns.adobe.com/flex/mx";
          s|Button {
               fontSize:16;
          }

          mx|VBox {
               color:red;
          }
     </fx:Style>
     <mx:VBox>
          <!-- This Spark button is red, and has a fontSize of 16. -->
          <s:Button label="Click Me, Too"/>
      </mx:VBox>
</s:Application>
```

You can also specify that a particular selector apply to mixed nested namespaces. This is common if you are using descendant selectors. You can exclude an identifier, in which case the declared namespace becomes the default namespace. The following example uses the default namespace:

```
<fx:Style>
    @namespace "library://ns.adobe.com/flex/spark";
    Button {
        color: #990000;
    }
</fx:Style>
```

For custom components, you can define your own style namespace; that style namespace must match the component's namespace. For example, components in the myComponents package use the style namespace "myComponents.*", as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/CustomComponentsNamespace.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:comps="myComponents.*">
    <fx:Style>
        @namespace comps "myComponents.*";

        comps|MyButton {
            color:green;
        }
    </fx:Style>
    <comps:MyButton label="Click Me"/>
</s:Application>
```

For custom components that are in the top level package, you can use an "*" for the namespace. However, you cannot use wildcard namespace prefixes, such as "*|" to match any namespace.

You can create your own namespace. This can be useful if you want to defines type selectors for custom components. The following example creates a new namespace for the classes in the mx.charts.chartClasses.* package. It then applies the new styles to just the DataTip class in that package:

```
<?xml version="1.0"?>
<!-- charts/DataTipStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="srv.send()"
    height="600">
    <fx:Declarations>
        <!-- View source of the following page to see the structure of the data that Flex uses 
in this example. -->
         <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
         <!-- To see data in an HTML table, go to 
http://aspexamples.adobe.com/chart_examples/expenses.aspx -->
    </fx:Declarations>
    <fx:Style>
        @namespace chartClasses "mx.charts.chartClasses.*";
        chartClasses|DataTip {
            fontFamily: "Arial";
            fontSize: 12;
            fontWeight:bold;
            fontStyle:italic;
        }
    </fx:Style>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel title="Bar Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:BarChart id="myChart"
            dataProvider="{srv.lastResult.data.result}"
            showDataTips="true">
            <mx:verticalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:verticalAxis>
            <mx:series>
                <mx:BarSeries
                    yField="month"
                    xField="profit"
                    displayName="Profit"/>
                <mx:BarSeries
                    yField="month"
                    xField="expenses"
                    displayName="Expenses"/>
            </mx:series>
        </mx:BarChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

About inheritance in CSS

If you set an inheritable style property on a parent container, its children inherit that style property. For example, if you define `fontFamily` as `Times` for a Panel container, all children of that container will also use `Times` for `fontFamily`, unless they override that property. If you set a noninheritable style on a parent container, only the parent container uses that style; the children do not use that style. For more information on inheritable style properties, see "About style inheritance" on page 1526.

In general, color and text styles are inheritable, regardless of which theme they are in (Spark or Halo) or how they are set (by using style sheets or the setStyle() method).

The following are exceptions to the rules of inheritance:

- If you use the `global` selector in a CSS style definition, Flex applies those style properties to all controls, regardless of whether the properties are inheritable. For more information, see "Using the global selector" on page 1536.
- The values set in type selectors apply to the target class as well as its subclasses, even if the style properties are not inheritable. For example, if you define a Group type selector, Flex applies all styles in that selector to Group and VGroup controls because VGroup is a subclass of Group.

In general, you should avoid using type selectors for commonly-used base classes like Group. Group is a class that Spark skins are based on. Setting styles on the Group type selector might cause unexpected results because the skins will have styles applied that you might not oridinarily expect to apply.

The following example illustrates this issue. All labels and button labels are yellow, even though only one label and one button is explicitly in a group. The reason is that the LabelSkin and ButtonSkin classes that define the skins for the Label and Button components, are based on the SparkSkin class. The SparkSkin class is a subclass of Group.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/TypeSelectorInheritance.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
          @namespace s "library://ns.adobe.com/flex/spark";
          @namespace mx "library://ns.adobe.com/flex/mx";
          s|Group {
               color:#FFCC33;
          }

     </fx:Style>

     <s:VGroup>
        <s:Button label="This Button is in a VGroup and its label is yellow."/>
        <s:Label text="This Label is in a VGroup and its text is yellow."/>
     </s:VGroup>
     <s:Button label="This Button is not in a Group, but it is still yellow."/>
     <s:Label text="This Label is not in a Group, but it is still yellow."/>
</s:Application>
```

CSS differences

There are some major differences in Flex between support of CSS and the CSS specification:

- Flex supports subclassing selectors. For example, if you specify a Box type selector, Flex applies the styles to all subclasses of Box, including VBox and HBox. This applies to all classes in the Flex framework's class hierarchy below UIComponent. In this case, UIComponent is considered a "stop class."
- Flex supports a subset of the style properties that are available in CSS. Flex controls also have unique style properties that are not defined by the CSS specification.
- Flex controls only support styles that are defined by the current theme. If a theme does not use a particular style, applying that style to a control or group of controls has no effect. For more information, see "About themes" on page 1561.
- Flex style sheets can define skins for controls using the Embed keyword. For more information, see "Skinning MX components" on page 1655.
- The CSS 3 wildcard namespace prefix syntax of *| that matches any namespace (including no namespace) is not supported. For more information, see "Using Cascading Style Sheets" on page 1504.
- The universal selector scoped to a particular CSS namespace is not supported. Namespaces are not known at runtime in Flex and as such the universal selector remains universal no matter what namespace it is scoped to in CSS.
- If you apply multiple class selectors to a component, the order of preference that the styles is applied is the order in which they appear in the `styleName` property's space-delimited list, not the order in which they are defined in the CSS block or external style sheet.

- The implementation of the CSS Media Query feature in Flex supports only the “screen” type. No other media types are supported.

About class selectors

Class selectors define a set of styles (or a class) that you can apply to any component. You define the style class, and then point to the style class using the `styleName` property of the component’s MXML tag. All components that are a subclass of the UIComponent class support the `styleName` property.

The following example defines a new style `myFontStyle` and applies that style to a Button component by assigning the Button to the `myFontStyle` style class:

```
<?xml version="1.0"?>
<!-- styles/ClassSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        .myFontStyle {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>
    <s:VGroup>
        <!-- This button has the custom style applied to it. -->
        <s:Button id="myButton" styleName="myFontStyle" label="Click Me"/>
        <!-- This button uses default styles. -->
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

You can apply multiple class selectors by separating them with a space in the `styleName` property. The order of preference that the styles is applied is the order in which they appear in the space-delimited list, not the order in which they are defined in the CSS block or external style sheet. The following example applies both class selectors to the first button:

```
<?xml version="1.0"?>
<!-- styles/MultipleStyleNames.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        .myFontStyle {
            fontSize: 22;
        }
        .myOtherFontStyle {
            color: #9933FF;
        }
    </fx:Style>
    <s:VGroup>
        <!-- This button has the custom style applied to it. -->
        <s:Button id="myButton"
            styleName="myFontStyle myOtherFontStyle"
            label="Click Me"/>
        <!-- This button uses default styles. -->
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

Class selector names must start with a period when you access them with the `getStyleDeclaration()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ClassSelectorStyleManager.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        .myFontStyle {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>
    <fx:Script>
        <![CDATA[
            public function changeStyles(e:Event):void {
                styleManager.getStyleDeclaration(".myFontStyle").setStyle("color",0x3399CC);
            }
        ]]>
    </fx:Script>
    <s:Button id="myButton" label="Click Here" styleName="myFontStyle"
click="changeStyles(event)"/>

</s:Application>
```

You do not precede the class selector with a period when you use the `styleName` property for inline styles.

About type selectors

Type selectors assign styles to all components of a particular type. When you define a type selector, you are not required to explicitly apply that style. Instead, Flex applies the style to all classes of that type. Flex also applies the style properties defined by a type selector to all subclasses of that type.

When you use type selectors, you are required to specify which namespace each type appears in with the `@namespace` directive.

The following example shows a type selector for Button controls:

```
<?xml version="1.0"?>
<!-- styles/TypeSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            fontSize: 15;
            color: #9933FF;
        }
    </fx:Style>
    <s:VGroup>
        <s:Button id="myButton" label="Click Me"/>
        <s:Button id="myButton2" label="Click Me"/>
    </s:VGroup>
</s:Application>
```

In this example, Flex applies the `color` style to all Spark Button controls in the current document, and all Button controls in all the child documents. In addition, Flex applies the `color` style to all subclasses of the Spark Button class.

You can set the same style declaration for multiple component types by using a comma-separated list of components. The following example defines style information for all Spark Button, TextInput, and Label components:

```
<?xml version="1.0"?>
<!-- styles/MultipleTypeSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button, s|TextInput, s|Label {
            fontStyle: italic;
            fontSize: 24;
        }
    </fx:Style>
    <s:Button id="myButton" label="Click Here"/>
    <s:Label id="l1" text="My Label"/>
    <s:TextInput id="ti1" text="Input text here"/>
</s:Application>
```

You can use multiple type selectors of the same name at different levels to set different style properties. In an external CSS file, you can set all Spark Button components to use the Blue color for the fonts, as the following example shows:

```
/* assets/SimpleTypeSelector.css */
@namespace s "library://ns.adobe.com/flex/spark";
s|Button {
    fontStyle: italic;
    color: #99FF00;
}
```

Then, in a local style declaration, you can set all Buttons to use the font size 10, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/TypeSelectorWithExternalCSS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style source="../assets/SimpleTypeSelector.css"/>
  <fx:Style>
     @namespace s "library://ns.adobe.com/flex/spark";
     s|Button {
        fontSize: 15;
     }
  </fx:Style>
  <s:Button id="myButton" label="Click Here"/>
</s:Application>
```

The local style declaration does not interfere with external style declarations. Flex applies only the style properties that you specified. The result of this example is that Spark Label controls that are children of the current document use Blue for the color and 10 for the font size.

All styles are shared across all documents in an application and across all applications that are loaded inside the same application. For example, if you load two SWF files inside separate tabs in an MX TabNavigator container, both SWF files share the external style definitions.

When you assign styles to a type selector, all subclasses of that selector’s type (the class) are affected by the styles. For example, if you create a type selector for Group, the styles are also applied to VGroup and HGroup instances because those classes are subclasses of Group.

Using compound selectors

You can mix class and type selectors to create compound style declarations. For example, you can define the color in a class selector and the font size in a type selector, and then apply both to a component:

```
<?xml version="1.0"?>
<!-- styles/CompoundSelectors.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
     @namespace s "library://ns.adobe.com/flex/spark";

     s|Label {
        fontSize: 10pt;
     }
     .myLabel {
        color: Blue;
     }
  </fx:Style>
  <s:Label styleName="myLabel" text="This Label is 10pt Blue"/>
</s:Application>
```

If you later remove one of the selectors (for example, call the StyleManager class' `clearStyleDeclaration()` method on the type or class selector), the other selector's style settings remain.

Using id selectors

Flex supports using an id selector. Flex applies styles to a component whose id matches the id selector in the CSS. To define an id selector, specify the id of the component with a pound sign (#) followed by the id string. The id selector can be qualified by the type or by itself.

The following example shows two ways to use an id selector:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/CSSIDSelectorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:HorizontalLayout/>
     </s:layout>

     <fx:Style>
          @namespace s "library://ns.adobe.com/flex/spark";
          @namespace mx "library://ns.adobe.com/flex/mx";
          s|Button {
               fontSize:16;
          }

          #myButton2 {
               color:red;
          }

          s|Button#myButton3 {
               color:blue;
          }
     </fx:Style>

     <s:Button id="myButton1" label="Click Me"/>
     <s:Button id="myButton2" label="Click Me, Too"/>
     <s:Button id="myButton3" label="Click Me, Three"/>
</s:Application>
```

Using descendant selectors

Descendant selectors are applied to components in a document, depending on their relationship to other components in the document. A descendant selector lets you apply styles to a component based on whether they descend (are children, grandchildren, or great grandchildren) from particular types of components.

When a component matches multiple descendant selectors, it adopts the style of the most closely-related ancestor. If there are nested descendant selectors, the component uses the styles defined by the selector that most closely matches its line of ancestors. For example, a Spark Button control within a VGroup within another VGroup matches a descendant selector for “s|VGroup s|VGroup s|Button” rather than a descendant selector for “s|VGroup s|Button”.

The following example shows four selectors. The first class selector applies to all Button instances. The first descendant selector applies to the second button, because that Button’s parent is a VGroup. The second descendant selector applies to the third Button, because that Button is a child of an HGroup and a VGroup. The last descendant select applies to the last Button, because that Button is a child of a VGroup within a VGroup.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/CSSDescendantSelectorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
          @namespace s "library://ns.adobe.com/flex/spark";
          @namespace mx "library://ns.adobe.com/flex/mx";
          s|Button {
               fontSize:16;
          }

          s|VGroup s|Button {
               color:red;
          }

          s|VGroup s|HGroup s|Button {
               color: blue;
          }
          s|VGroup s|VGroup s|Button {
               color: green;
          }
     </fx:Style>
     <!-- This button has a fontSize of 16. -->
     <s:Button label="Click Me"/>
     <s:VGroup>
          <!-- This button is red, and has a fontSize of 16. -->
          <s:Button label="Click Me, Too"/>
      </s:VGroup>
     <s:VGroup>
          <s:HGroup>
               <!-- This button is blue, and has a fontSize of 16. -->
               <s:Button label="Click Me, Also"/>
          </s:HGroup>
     </s:VGroup>
     <s:VGroup>
          <s:VGroup>
               <!-- This button is green, and has a fontSize of 16. -->
               <s:Button label="Click Me, Click Me!"/>
          </s:VGroup>
     </s:VGroup>
</s:Application>
```

Styles are applied only to components that appear in the display list. Descendant selectors are only applied if the ancestors also appear in the display list.

Descendant selectors work with all classes that implement the IStyleClient interface.

A descendant selector applies to a component as long as any parent class of that component matches the descendant selector. The following example shows that two Spark Button controls inherit the styles of the Group descendant selector, because their parents (VGroup and HGroup) are both subclasses of Group.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/CSSDescendantSelectorExample2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComps="*">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>
     <fx:Style>
          @namespace s "library://ns.adobe.com/flex/spark";
          @namespace mx "library://ns.adobe.com/flex/mx";

          s|Group s|Button {
               color:red;
          }
     </fx:Style>
     <s:VGroup>
          <!-- This button is red, because VGroup is a subclass of Group. -->
          <s:Button label="Click Me"/>
     </s:VGroup>
     <s:HGroup>
          <!-- This button is also red, because HGroup is also a subclass of Group. -->
          <s:Button label="Click Me, Too"/>
     </s:HGroup>
</s:Application>
```

In general, you should avoid using the Group type selector in CSS. This is because all Spark skins use the SparkSkin class, which is a subclass of Group. As a result, Spark skins will inherit style properties from the Group type selectors, which might produce unexpected results.

Nested descendant selectors that set inheritable style properties also work for subcomponents. For example, a Label is a subcomponent of a Button control. It is responsible for rendering the text for the button’s label text.

The following example shows that the nested descendant selectors Button Label apply to a Button for inheritable style properties:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/SubComponentDescendantSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
          @namespace s "library://ns.adobe.com/flex/spark";
          @namespace mx "library://ns.adobe.com/flex/mx";
          s|Button s|Label {
               backgroundColor:#FFCC33;
          }
     </fx:Style>

     <!-- This button has a fontSize of 16. -->
     <s:Button label="Click Me"/>
</s:Application>
```

Using pseudo selectors

Pseudo selectors let you apply styles to a component when that component is in a particular state. The following example uses a pseudo selector to change the appearance of the button when it is in the up, down, and over states:

```
<?xml version="1.0"?>
<!-- styles/PseudoSelectorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:custom="*">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";

        s|Button:up {
            chromeColor: black;
            color: #FFFFFF;
        }
        s|Button:over {
            chromeColor: gray;
            fontWeight: "bold";
            color: #FFFFFF;
        }
        s|Button:down {
            chromeColor: blue;
            fontWeight: "bold";
            color: #FFFF66;
        }
    </fx:Style>
    <s:Button label="Click Me" x="10" y="35"/>
</s:Application>
```

For more information about states, see “View states” on page 1847.

About selector precedence

In general, the following rules apply when determining which styles are applied:

- Local styles override global styles (a style set inline takes precedence over a selector)
- When styles have equivalent precedence, the last one applied takes precedence
- The more specific style takes precedence over the more general style

The rules for determining precedence are complicated. The Flex rules follow the CSS rules that are defined by the W3C.

For more information, see CSS3 Selectors W3C specification.

Class selectors take precedence over type selectors. In the following example, the text for the first Button control (with the class selector) is red, and the text of the second Button control (with the implicit type selector) is yellow:

```
<?xml version="1.0"?>
<!-- styles/SelectorPrecedence.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        .myclass {
            color: Red;
        }
        s|Button {
            fontSize: 10pt;
            color: Yellow;
        }
    </fx:Style>
    <s:VGroup width="500" height="200">
        <s:Button styleName="myclass" label="Red Button"/>
        <s:Button label="Yellow Button"/>
    </s:VGroup>
</s:Application>
```

The font size of both buttons is 10. When a class selector overrides a type selector, it does not override all values, just those that are explicitly defined.

Type selectors apply to a particular class, as well as its subclasses and child components. In the following example, the `color` property for a VGroup control is `blue`. This means that the `color` property for the Button and Label controls, which are direct children of the VGroup control, is blue.

```
<?xml version="1.0"?>
<!-- styles/BasicInheritance.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|VGroup {
            color:blue
        }
    </fx:Style>
    <s:VGroup width="500" height="200">
        <s:Label text="This is a Label control."/>
        <s:Button label="Click Me"/>
    </s:VGroup>
</s:Application>
```

If the same style property is applied in multiple type selectors that apply to a class, the closest type to that class takes precedence. For example, the MX VBox class is a subclass of Box, which is a subclass of Container. If there were Box and Container type selectors rather than a VBox type selector, then the value of the VBox control's `color` property would come from the Box type selector rather than the Container type selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ContainerInheritance.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Container {
            color:red
        }
        mx|Box {
            color:green
        }
    </fx:Style>
    <mx:VBox width="500" height="200">
        <mx:Label text="This is a green label."/>
        <mx:Button label="Click Me"/>
    </mx:VBox>
</s:Application>
```

Not all style properties are inheritable. For more information, see "About style inheritance" on page 1526.

Embedding resources in style sheets

You can use embedded resources in your `<fx:Style>` blocks. This is useful for style properties such as `backgroundImage`, which you can apply to an embedded resource such as an image file. The following example embeds an image in CSS:

```
<?xml version="1.0"?>
<!-- styles/EmbedInCSS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        .style1 {
            backgroundImage: Embed("../assets/butterfly.gif");
            backgroundAlpha: .2;
        }
    </fx:Style>

    <s:Panel title="BorderContainer Component Example"
            width="75%" height="75%"
            horizontalCenter="0" verticalCenter="0">
        <s:BorderContainer
            left="10" right="10" top="10" bottom="10"
            styleName="style1">
            <s:layout>
                <s:VerticalLayout
                    paddingLeft="5" paddingRight="5"
                    paddingTop="5" paddingBottom="5"/>
            </s:layout>
            <s:HGroup>
                <s:Button label="Button 1"/>
                <s:Button label="Button 2"/>
                <s:Button label="Button 3"/>
                <s:Button label="Button 4"/>
            </s:HGroup>
            <s:HGroup>
                <s:Button label="Button 5"/>
                <s:Button label="Button 6"/>
                <s:Button label="Button 7"/>
                <s:Button label="Button 8"/>
            </s:HGroup>
            <s:HGroup>
                <s:Button label="Button 9"/>
                <s:Button label="Button 10"/>
                <s:Button label="Button 11"/>
                <s:Button label="Button 12"/>
            </s:HGroup>
        </s:BorderContainer>
    </s:Panel>
</s:Application>
```

For graphical Halo skins, you use the `Embed` statement directly in the style sheet, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesTypeSelector.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";

        mx|Button {
            overSkin: Embed("../assets/orb_over_skin.gif");
            upSkin: Embed("../assets/orb_up_skin.gif");
            downSkin: Embed("../assets/orb_down_skin.gif");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

For programmatic Halo skins, you use the `ClassReference` statement, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatesSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            overSkin: ClassReference("ButtonStatesSkin");
            upSkin: ClassReference("ButtonStatesSkin");
            downSkin: ClassReference("ButtonStatesSkin");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

For more information about using the `Embed` keyword, see “Embedding assets” on page 1699.

For Spark controls, you edit the skin class to add images and behaviors for certain states. For more information about skinning Spark components, see “Spark Skinning” on page 1602.

For more information about skinning MX components, see “Skinning MX components” on page 1655.

Using document properties in CSS

You can use document properties in your CSS with the `PropertyReference` keyword. The property must be public, or it must be defined in the same document as the CSS. For example, you can use a private variable in an imported CSS file, as long as you imported that CSS file in the same document in which the variable is declared.

If you change the value of the document property, the component’s style property is not updated, even if the document property is bindable. Changing the value of the document property will not update the control’s appearance because styles are not reapplied unless you explicitly call the `setStyle()` method or trigger a reapplication of the styles in some other way.

The following example defines the mySize variable and uses it in the CSS:

```
<?xml version="1.0"?>
<!-- styles/PropertyReferenceExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <!-- You can declare a property using this method, too: -->
  <!--
  <fx:Declarations>
    <fx:Number id="mySize">20</fx:Number>
  </fx:Declarations>
  -->

  <fx:Script>
    [Bindable]
    private var mySize:Number = 20;
  </fx:Script>

  <fx:Style>
     @namespace s "library://ns.adobe.com/flex/spark";
     s|Button {
        fontSize: PropertyReference("mySize");
     }
  </fx:Style>
    <!--Notice that when you click the button, the value of mySize increases,
         but the size of the font on the button's label does not. Style properties
         must be explicitly set, even if the property is bindable. -->
  <s:Button id="myButton" label="{mySize.toString()}" click="mySize+=2"/>
</s:Application>
```

Using media queries

Flex includes partial support for CSS media queries. In Flex, media queries let you conditionally apply styles based on the DPI and OS of the target device. You typically use this feature if you are developing for mobile or tablet devices that have different target DPIs. For example, if a tablet has a DPI of 300, then you can set the font size to one value. If a tablet has a DPI of 200, then you can set the font size to a different value.

Media queries use the values of the `application-dpi` and `os-platform` CSS properties to determine which styles to apply.

The syntax for using media queries is as follows:

```
@media [media_type] [application-dpi:180|240|320] [and|not|only]
    [os-platform:"Android"|"IOS"|"Macintosh"|"Windows"|"Linux"]
```

The only supported value for the `media_type` property is `screen`. As a result, you do not need to specify the media type.

If you do not specify a value for the `application-dpi` or `os-platform` properties, then all are assumed. You can specify one or more resolutions or platforms by comma-separating their values.

The following example changes the font size and color based on the value of the `applicationDPI` property:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/MediaQueryExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
     xmlns:s="library://ns.adobe.com/flex/spark"
     applicationDPI="320"
     creationComplete="initApp()">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        s|TextArea {
            fontSize: 12;
        }
      @media (application-dpi: 160) and (os-platform: "Windows"), (os-platform: "Macintosh"),
(os-platform: "Linux") {
            s|TextArea {
                fontSize: 12;
                color: red;
            }
        }
      @media (application-dpi: 240) and (os-platform: "Windows"), (os-platform: "Macintosh"),
(os-platform: "Linux") {
            s|TextArea {
                fontSize: 18;
                color: green;
            }
        }
      @media (application-dpi: 320) and (os-platform: "Windows"), (os-platform: "Macintosh"),
(os-platform: "Linux") {
            s|TextArea {
                fontSize: 24;
                color: blue;
            }
        }
    </fx:Style>

    <fx:Script>
        private function initApp():void {
            myTextArea.text = "OS: " + Capabilities.os + "\n"
                + "MFR: " + Capabilities.manufacturer + "\n"
                + "DPI: " + applicationDPI;
        }
    </fx:Script>

    <s:TextArea id="myTextArea" width="100%"/>

</s:Application>
```

For more information and examples on using media queries in applications, see Support multiple screen sizes and DPI values in a mobile application.

About style inheritance

If you define a style in only one place in a document, Flex uses that definition to set a property's value. However, an application can have several style sheets, local style definitions, external style properties, and style properties set directly on component instances. In such a situation, Flex determines the value of a property by looking for its definition in all these places in a specific order.

Lower-level styles take precedence over higher-level or external styles. If you set a style on an instance, and then set the style globally, the global style does not override the local style, even if you set it after you set the local style.

Style inheritance order

The order in which Flex looks for styles is important to understand so that you know which style properties apply to which controls.

Flex looks for a style property that was set inline on the component instance. If no style was set on the instance using an inline style, Flex checks if a style was set using an instance's setStyle() method. If it did not directly set the style on the instance, Flex examines the `styleName` property of the instance to see if a style declaration is assigned to it.

If you did not assign the `styleName` property to a style declaration, Flex looks for the property on type selector style declarations. If there are no type selector declarations, Flex checks the `global` selector. If all of these checks fail, the property is undefined, and Flex applies the default style.

In the early stages of checking for a style, Flex also examines the control's parent container for style settings. If the style property is not defined and the property is inheritable, Flex looks for the property on the instance's parent container. If the property isn't defined on the parent container, Flex checks the parent's parent, and so on. If the property is not inheritable, Flex ignores parent container style settings.

The order of precedence for style properties, from first to last, is as follows:

- Inline
- Class selector
- Type selectors (most immediate class takes precedence when multiple selectors apply the same style property)
- Ancestor class's type selector
- Parent chain (inheriting styles only)
- Theme defaults.css file
- `global` selector

If you later call the `setStyle()` method on a component instance, that method takes precedence over all style settings, including inline.

Style definitions in `<fx:Style>` tags, external style sheets, and the defaults.css style sheet follow an order of precedence. The same style definition in defaults.css is overridden by an external style sheet that is specified by an `<fx:Style source="`*stylesheet*`"/>` tag, which is overridden by a style definition within an `<fx:Style>` tag.

The following example defines a type selector for Panel that sets the `fontFamily` property to `Times` and the `fontSize` property to 24. As a result, all controls inside the Panel container, as well as all subclasses such as Button and TextArea, inherit those styles. However, button2 overrides the inherited styles by defining them inline. When the application renders, button2 uses Arial for the font and 12 for the font size.

```
<?xml version="1.0"?>
<!-- skins/MoreContainerInheritance.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Panel {
             fontFamily: Times, "_serif";
             fontSize: 24;
          }
  </fx:Style>
  <s:Panel title="My Panel">
     <s:Button id="button1" label="Button 1"/>
     <s:Button id="button2" label="Button 2" fontFamily="Arial" fontSize="12"/>
     <s:TextArea text="Flex has is own set of style properties which are
        extensible so you can add to that list when you create a custom
        component." width="425" height="400"/>
  </s:Panel>
</s:Application>
```

Subcomponent styles

Some Flex controls are made up of other components. For example, the DateField control includes a DateChooser subcomponent, the calendar that pops up when you click on the DateField's icon. The DateChooser subcomponent itself contains Button subcomponents for navigation and TextField subcomponents for labels.

The following table lists some of the most commonly used Spark components and their subcomponents:

Spark component	Subcomponents
Button	Label
ComboBox	Button, Label
HScrollBar/VScrollBar	Button
HSlider/VSlider	Button, Label
NumericStepper	Button, TextInput
RadioButton	Label
TextArea	RichEditableText
TextInput	RichEditableText
TitleWindow	Button, Label
VideoPlayer	Button, Label

Inheritable styles are passed from the parent control to the subcomponent. These include text styles like `color` and `textDecoration`. If you set the `color` style property on a DateField control, Flex applies that `color` to the text in the DateChooser subcomponent, too.

If a style property is explicitly set on the subcomponent (such as in its skin class), then you cannot override it by setting it in CSS. You must edit the skin class or override the style property value in some other way. For example, the Spark HSlider control has a Label subcomponent that defines the appearance of the slider's labels. The `color` style property is set on this Label control explicitly in the HSliderSkin class:

```
<!-- From HSliderSkin.mxml -->
<s:Label id="labelDisplay" text="{data}"
    horizontalCenter="0" verticalCenter="1"
    left="5" right="5" top="5" bottom="5"
    textAlign="center" verticalAlign="middle"
    fontWeight="normal" color="white" fontSize="11">
</s:Label>
```

If you define a Label type selector and specify the color, your CSS setting does not override the explicit setting in the HSlider control's skin class. In this case, you must create a custom skin class for the HSlider control to change the color of the label's text.

If you do not want an inheritable style property to be applied to the subcontrol, you can override the parent's style by defining a custom class selector. For example, to apply styles to the subcomponents of a ComboBox control, you can use the `dropdownStyleName` or `textInputStyleName` style properties to define custom selectors.

Most controls that have subcomponents have custom class selectors that apply styles to their subcomponents. In some cases, controls have multiple custom class selectors so that you can set style properties on more than one subcomponent.

The following example sets the `color` style property on the DateField control. To prevent this color from being applied to the DateChooser subcomponent, the DCStyle custom class selector overrides the value of the `color` style property. This custom class selector is applied to the DateField control with the `dateChooserStyleName` property.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/SubComponentStylesSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

    <fx:Style>
        .DCStyle {
            color:blue;
        }
    </fx:Style>
    <s:VGroup>
        <s:Label text="Overrides the color property of the subcontrol:"/>
        <mx:DateField
            id="dateField1"
            yearNavigationEnabled="true"
            color="red"
            dateChooserStyleName="DCStyle"/>
    </s:VGroup>
    <mx:HRule width="200" height="1"/>
    <s:VGroup>
        <s:Label text="Applies the color property to the subcontrol:"/>
        <mx:DateField
            id="dateField2"
            yearNavigationEnabled="true"
            color="red"/>
    </s:VGroup>
</s:Application>
```

Noninheritable style properties are not passed from the parent component to the subcomponent. In some cases, the parent control provides a property that lets you access the subcomponent. For example, to access the RichEditableText control that is a subcontrol of the Spark TextArea and TextInput controls, you use the `textDisplay` property.

There are some exceptions to noninheritable styles. For example, the `verticalAlign`, `lineBreak`, and `paddingBottom/Left/Right/Top` style properties are noninheritable. You can set these properties, which are defined on the RichEditableText control, directly on the TextArea control because the TextAreaSkin class passes them through to the subcomponent.

The following example sets styles on the RichEditableText subcomponent with the `textDisplay` property, and also sets the values of some noninheriting style properties that are not normally accessible:

```
<?xml version="1.0"?>
<!-- styles/TextAreaStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            /*
            Non-inheriting styles you must set on textDisplay:
                columnCount
                columnGap
                columnWidth

            Non-inheriting styles that you can set on TextArea because
            they are passed to the subcomponent through the TextAreaSkin class:
                lineBreak
                paddingTop/Bottom/Left/Right
                verticalAlign
            */
            import spark.components.RichEditableText;

            private function initApp():void {
                RichEditableText(ta1.textDisplay).setStyle("columnCount", 3);
                RichEditableText(ta1.textDisplay).setStyle("columnWidth", 100);
                RichEditableText(ta1.textDisplay).setStyle("columnGap", 15);
            }
        ]]>
    </fx:Script>
    <s:TextArea id="ta1" height="100" width="400" verticalAlign="bottom" paddingBottom="20">
        This is a text area control. Because the text rendering is done by a RichEditableText
subcontrol,
      you have to use the textDisplay property to set the values of some non-inheriting styles.
       Other non-inheriting styles are defined in the skin class and are passed through to the
        subcomponent.
        For inheriting styles, they are inherited by the RichEditableText subcontrol.
    </s:TextArea>
</s:Application>
```

Some controls use filters to determine which style properties are passed to subcontrols. For example, if you customize the value of the `cornerRadius` property on a MX DateChooser control, the property does not affect the buttons that are subcomponents of the calendar. To pass that value to the subcomponent in MX components, you can modify the control's filter. Filters specify which style properties to pass to their subcomponents. A filter is an Array of objects that define the style properties that the parent control passes through to the subcomponent. Inheritable style properties are always passed; they cannot be filtered.

Most MX components with subcomponents have at least one filter. For example, the ComboBox subcontrol has a `dropDownStyleFilters` property that defines which style properties the ComboBox passes through to the drop down List subcomponent.

Some MX controls with subcomponents have multiple filters. For example, the DateChooser control has separate filters for each of the buttons on the calendar: the previous month button (`prevMonthStyleFilters`), the next month button (`nextMonthStyleFilters`), the previous year button (`prevYearStyleFilters`), and the next year button (`nextYearStyleFilters`).

The filters properties are read-only, but you can customize them by subclassing the control and adding or removing objects in the filter Array.

The following example includes two DateField controls. The first DateField control does not use a custom filter. The second DateField control is a custom class that uses two custom filters (one for the properties of the next month button and one for the properties of the previous month button).

```
<?xml version="1.0" encoding="utf-8"?>
<!-- versioning/StyleFilterOverride.mxml -->
<!-- Compile this example by setting the theme argument to use the Halo theme. -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="*">
     <s:layout>
        <s:HorizontalLayout/>
     </s:layout>
    <s:VGroup>
        <s:Label width="200"
           text="Standard DateChooser control. Does not pass the cornerRadius property to the
button subcomponents:"/>
        <mx:DateChooser cornerRadius="10"/>
    </s:VGroup>
    <s:VGroup>
        <s:Label width="200"
            text="Custom DateChooser control. Passes the cornerRadius property to the button
subcomponents:"/>
        <comps:MyDateChooser cornerRadius="10"/>
    </s:VGroup>
</s:Application>
```

To compile this example, you cannot use the Spark theme. You must set `theme` compiler option to the Halo theme file. To use a background image on a container with the Spark theme, you must create a custom skin class.

The following class extends the DateChooser class and defines custom filters Arrays for two of the button subcomponents:

```
// styles/MyDateChooser.as
package {
    import mx.controls.DateChooser;

    public class MyDateChooser extends DateChooser {
        private static var myNextMonthStyleFilters:Object = {
             "highlightAlphas" : "highlightAlphas",
             "nextMonthUpSkin" : "nextMonthUpSkin",
             "nextMonthOverSkin" : "nextMonthOverSkin",
             "nextMonthDownSkin" : "nextMonthDownSkin",
             "nextMonthDisabledSkin" : "nextMonthDisabledSkin",
             "nextMonthSkin" : "nextMonthSkin",
             "repeatDelay" : "repeatDelay",
             "repeatInterval" : "repeatInterval",
             "cornerRadius" : "cornerRadius"  // This property is not normally included.
        }

       override protected function get nextMonthStyleFilters():Object {
            return myNextMonthStyleFilters;
       }

       private static var myPrevMonthStyleFilters:Object = {
            "highlightAlphas" : "highlightAlphas",
            "prevMonthUpSkin" : "prevMonthUpSkin",
            "prevMonthOverSkin" : "prevMonthOverSkin",
            "prevMonthDownSkin" : "prevMonthDownSkin",
            "prevMonthDisabledSkin" : "prevMonthDisabledSkin",
            "prevMonthSkin" : "prevMonthSkin",
            "repeatDelay" : "repeatDelay",
            "repeatInterval" : "repeatInterval",
            "cornerRadius" : "cornerRadius"
        }

        override protected function get prevMonthStyleFilters():Object {
            return myPrevMonthStyleFilters;
        }
    }
}
```

The custom filters each include the following additional entry in the Array:

```
"cornerRadius" : "cornerRadius"
```

The `cornerRadius` property is not normally listed in the `nextMonthStyleFilters` and `myPrevMonthStyleFilters` Arrays. By adding it to these filter Arrays, you ensure that the property is passed from the parent control to the subcontrol, and is applied to the previous month and next month buttons.

You can also use filters to exclude properties that are normally passed through the subcomponent. You do this by removing those properties from the filter Array in the subclass.

Inheritance exceptions for styles

Not all styles are inheritable, and not all styles are supported by all components and themes. In general, color and text styles are inheritable, regardless of how they are set (using CSS or style properties). All other styles are not inheritable unless otherwise noted.

A style is inherited only if it meets the following conditions:

- The style is inheritable. You can see a list of inherited style for each control by viewing that control's entry in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. You can programmatically determine if a style is inheritable using the static `isInheritingStyle()` or `isInheritingTextFormatStyle()` methods on the StyleManager class. You can access the top-level StyleManager by using the `styleManager` property of the Application object.
- The style is supported by the theme. To determine if a style property is supported by the theme you are using, view the style's entry in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. If the style property is limited to a theme, the `Theme` property will appear in the style's description, with the name of the supported theme next to it. If the style requires a different theme, you can use the `theme` compiler option to change the theme. If the *ActionScript 3.0 Reference for the Adobe Flash Platform* does not specify a theme for a particular style property, then that property should work with all themes.
- The style is supported by the control. For information about which controls support which styles, see the control's description in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.
- The style is set on the control's parent container or the container's parent. A style is not inherited from another class, unless that class is a parent container of the control, or a parent container of the control's parent container. (The exception to this condition is if you use type selectors to apply the style property. In that case, Flex applies properties of the class's type selector, as well as any properties set in the base class's type selector.)
- The style is not overridden at a lower level. For example, if you define a style type selector (such as `Button { color:red }`), but then set an instance property on a control (such as `<mx:Button color="blue"/>`), the type selector style will not override the style instance property even if the style is inheritable.

You can apply noninheritable styles to all controls by using the `global` selector. For more information, see "Using the global selector" on page 1536.

About supported styles

All themes support the inheritable and noninheritable text styles, but not all styles are supported by all themes. If you try to set a style property on a control but the current theme does not support that style, Flex does not apply the style.

To determine if a style property is supported by the theme you are using, view the style's entry in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. If the style property is limited to a theme, the Theme property will appear in the style's description, with the name of the supported theme next to it. If there is no theme specified for a particular style property, then the style property should be supported by all themes.

Some styles are only used by skins in the theme, while others are used by the component code itself.

The display text of components is not skinnable, so support for text styles is theme-independent.

For more information, "About themes" on page 1561.

About the themeColor property

Many assets in the Halo theme support a property called `themeColor`. You can set this property on the MX Application tag, and the color is applied throughout the application on component assets, such as the MX Button control's border, the headers of an Accordion control, and the default shading of a ToolTip control's background.

In addition to color values such as 0xCCCCCC (for silver) or 0x0066FF (for blue), the following values for the `themeColor` property are valid:

- `haloOrange`
- `haloBlue`
- `haloSilver`

- haloGreen

The default value is `haloBlue`. The following example sets the value of `themeColor` to `haloOrange`:

```
<?xml version="1.0"?>
<!-- styles/ThemeColorExample.mxml -->
<!-- Compile this example by setting the theme compiler argument to Halo.swc. -->
<mx:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    themeColor="haloOrange">

    <fx:Script>
        <![CDATA[
            import mx.core.FlexGlobals;
            import mx.collections.ArrayCollection;
            [Bindable]
            public var themes:ArrayCollection = new ArrayCollection(
                [ "haloOrange", "haloBlue", "haloSilver", "haloGreen"]);

            private function closeHandler(e:Event):void {
                FlexGlobals.topLevelApplication.setStyle("themeColor",
ComboBox(e.target).selectedItem);
            }
        ]]>
    </fx:Script>
    <mx:ComboBox dataProvider="{themes}" width="150" close="closeHandler(event);"/>
    <mx:Button id="myButton" label="Click Me" toolTip="Click me"/>
</mx:Application>
```

To compile this example, you must use the Halo theme. For information on using themes, see “Using themes” on page 1562.

To achieve functionality similar to the themeColor property in a Spark application, you can use the `chromeColor` style property. This property is supported only by the Spark theme.

Using external style sheets

Flex supports external CSS style sheets. You can declare the location of a local style sheet or use the external style sheet to define the styles that all applications use. To apply a style sheet to the current document and its child documents, use the `source` property of the `<fx:Style>` tag.

***Note:** You should try to limit the number of style sheets used in an application, and set the style sheet only at the top-level document in the application (the document that contains the `Application` tag). If you set a style sheet in a child document, unexpected results can occur.*

The following example points to the MyStyleSheet.css file in the *flex_app_root*/assets directory:

```
<?xml version="1.0"?>
<!-- styles/ExternalCSSExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style source="../assets/SimpleTypeSelector.css"/>
    <s:Button id="myButton" label="Click Me"/>
</s:Application>
```

The value of the `source` property is the URL of a file that contains style declarations. When you use the `source` property, the contents of that `<fx:Style>` tag must be empty. You can use additional `<fx:Style>` tags to define other styles. Do not add `<fx:Style>` tags to your included file; it should follow standard CSS file syntax.

The external style sheet file can contain both type and class selectors.

If you are using Adobe® Flash® Builder™, you can generate style sheets from existing component definitions by using the Convert to CSS button in the Style pop-up menu in the Flex Properties view. You can export a custom style sheet for a single component type or class of components, or use that component's style properties to create a global style sheet.

You can also compile CSS files into SWF files and load them at run time. For more information, see "Loading style sheets at run time" on page 1547.

You can specify a CSS file as an argument to the `source-path` compiler argument. This lets you toggle among style sheets with that compiler argument.

About the default style sheets

Flex includes defaults.css style sheets in the following SWC files:

- spark.swc (Spark theme style sheet)
- frameworks.swc (default framework style sheet)

All applications use the framework style sheet. All Spark applications use the Spark style sheet, as well, or another style sheet if a new theme is applied.

The defaults.css style sheet in the frameworks.swc file applies default styles to all components. It uses a combination of global CSS style settings and embedded symbols from the Assets.swf file to apply default styles to your applications.

The defaults.css style sheet in the spark.swc file applies default styles to the Spark components. It provides the look and feel of the Spark theme. The Spark default style sheet is much simpler than the framework style sheet. It applies skin classes to the Spark components, and sets a limited number of style properties on the global selector. For more information about themes, see "About themes" on page 1561.

The framework style sheet is applied first, followed by the Spark style sheet. The result is that the Spark style sheet takes precedence where overlapping style definitions occur because the styles are applied after the framework styles.

Flex implicitly loads the default style sheet and Spark theme style sheet during compilation. You can explicitly point to other files by using the `defaults-css-url` compiler option. You can also rename the defaults.css files or remove them from the SWC files to disable them. You typically do not edit the default style sheets in the SWC files (and then recompile those SWC files) to change the default style values. Instead, you set new values on the global selector or create a new theme. For more information, see "Using the global selector" on page 1536.

The default style sheets define the look and feel for all components. If you apply additional themes or CSS files to your application, Flex still uses the styles in default style sheets, but only for the properties that your custom styles do not override. To completely eliminate the default styles from Flex, you must remove or override all styles defined in default style sheets.

Flex also includes other style sheets that let you apply a theme quickly and easily. For more information, see "About the included theme files" on page 1565.

Using local style definitions

The `<fx:Style>` tag contains style sheet definitions that adhere to the CSS syntax. These definitions apply to the current document and all children of the current document. You must also specify a namespace in the `<fx:Style>` tag if you use type selectors that might have ambiguous names.

The `<fx:Style>` tag uses the following syntax to define local styles:

```
<fx:Style>
    @namespace namespace_identifiernamespace_string;
    selector_name {
        style_property: value;
        [...]
    }
</fx:Style>
```

The following example defines a class and a type selector in the `<fx:Style>` tag:

```
<?xml version="1.0"?>
<!-- styles/CompoundLocalStyle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        .myFontStyle {
            fontSize: 15;
            color: #9933FF;
        }
        s|Button {
            fontStyle: italic;
        }
    </fx:Style>
    <s:Button id="myButton" styleName="myFontStyle" label="Click Me"/>
</s:Application>
```

Using the Application type selector

The Application container is the top-most container in an application. Styles defined on the Application type selector that are inheritable are inherited by all of the container's children as well as the container's subclasses. Styles that are not inheritable are only applied to the Application container itself and not its children.

Inheritable styles can be overriden at a lower level; for example, if you set the color style property on a container that has another container and a Button as children, but then set the color on the inner container, the Button control inherits the color from the inner container.

To use CSS to apply a noninheritable style globally, you can use the `global` selector. For more information, see "Using the global selector" on page 1536.

Use the following syntax to define styles for the Application type selector:

```
<fx:Style>
    @namespace_declaration
    namespace|Application { style_definition }
</fx:Style>
```

You can use the MX Application type selector to set the background image and other display settings that define the way the application appears in a browser. You can only do this if you apply the Halo theme to the application when you compile it. The following Halo example aligns the application to the left, removes margins, and sets the background image to be empty:

```
<?xml version="1.0"?>
<!-- styles/ApplicationTypeSelector.mxml -->
<!-- Compile this with theme=halo.swc -->
<mx:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Application {
            paddingLeft: 0px;
            paddingRight: 0px;
            paddingTop: 0px;
            paddingBottom: 0px;
            horizontalAlign: "left";
            backgroundColor: #FFFFFF; /* Change color of background to white. */
            backgroundImage: " "; /* The empty string sets the image to nothing. */
        }
    </fx:Style>
    <mx:Button id="myButton" styleName="myFontStyle" label="Click Me"/>
</mx:Application>
```

When the background image is set to the empty string, Flex does not draw the default gray gradient for an MX Application.

For the Spark Application class, you edit its skin class to set a background image and set padding properties. You can set the background color on the Spark Application class with the `backgroundColor` style property, just as you would set it on the MX Application class. For more information about skinning Spark components, see “Spark Skinning” on page 1602.

You can programmatically define values in the Application type selector using the StyleManager class. You can access the top-level StyleManager by using the `styleManager` property of the Application object. For more information, see “Using the StyleManager class” on page 1537.

Using the global selector

Flex includes a `global` selector that you can use to apply styles to all controls. Properties defined by a `global` selector apply to every control unless that control explicitly overrides it. Because the `global` selector is like a type selector, you do not preface its definition with a period in CSS.

The following example defines `fontSize` and `textDecoration` to the `global` selector:

```
<?xml version="1.0"?>
<!-- styles/GlobalTypeSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        global {
            fontSize:22;
            textDecoration: underline;
        }
    </fx:Style>
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>
    <s:Button id="myButton" label="Click Me"/>
    <s:Label id="myLabel" text="This is a Label control."/>
</s:Application>
```

You can also use the `getStyleDeclaration()` method to apply the styles with the global selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/GlobalTypeSelectorAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

    <fx:Script><![CDATA[
        public function initApp(e:Event):void {
            styleManager.getStyleDeclaration("global").setStyle("fontSize", 22);
          styleManager.getStyleDeclaration("global").setStyle("textDecoration", "underline");
        }
    ]]></fx:Script>
    <s:Button id="myButton" label="Click Me"/>
    <s:Label id="myLabel" text="This is a Label control."/>
</s:Application>
```

Class selectors, type selectors, and inline styles all override the `global` selector.

Using the StyleManager class

The StyleManager class lets you access class selectors and type selectors in ActionScript. It also lets you apply inheritable and noninheritable properties globally. Using the StyleManager, you can define new CSS style declarations and apply them to controls in your applications.

You can access the top-level StyleManager by using the `styleManager` property of the Application object.

Setting styles with the StyleManager

To set a value using the StyleManager, use the following syntax:

```
styleManager.getStyleDeclaration(style_name).setStyle("property", value);
```

The `styleManager` property refers to the top level StyleManager object for the application.

The *`style_name`* can be the literal `global`, a type selector such as Button or TextArea, or a class selector that you define in either the `<fx:Style>` tag or an external style sheet. Global styles apply to every object that does not explicitly override them.

The `getStyleDeclaration()` method is useful if you apply a noninheritable style to many classes at one time. This property refers to an object of type CSSStyleDeclaration. Type selectors and external style sheets are assumed to already be of type CSSStyleDeclaration. Flex internally converts class selectors that you define to this type of object.

The following examples illustrate applying style properties to the `Button`, `myStyle`, and `global` style names:

```
<?xml version="1.0"?>
<!-- styles/UsingStyleManager.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)">
    <fx:Style>
        .myStyle {
            color: red;
        }
    </fx:Style>
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>
    <fx:Script><![CDATA[
        /* To get a reference to the top-level StyleManager, use the Application object's
           styleManager property. */

        public function initApp(e:Event):void {
            /* Type selector; applies to all Buttons and subclasses of Button. */

styleManager.getStyleDeclaration("spark.components.Button").setStyle("fontSize",24);
            /* Class selector; applies to controls using the style
               named myStyle. Note that class selectors must be prefixed
               with a period. */
            styleManager.getStyleDeclaration(".myStyle").setStyle("color",0xCC66CC);
            /* Global style: applies to all controls. */
            styleManager.getStyleDeclaration("global").setStyle("fontStyle","italic");
        }
    ]]></fx:Script>
    <s:Button id="myButton" label="Click Me" styleName="myStyle"/>
    <s:Label id="myLabel" text="This is a Label control." styleName="myStyle"/>
</s:Application>
```

***Note:** If you set either an inheritable or noninheritable style to the `global` style, Flex applies it to all controls, regardless of their location in the hierarchy.*

Accessing selectors with the StyleManager

You can access a list of all the class and type selectors that are currently registered with the StyleManager by using the StyleManager's `selectors` property. You can access the top-level StyleManager by using the `styleManager` property of the Application object. The selectors listed include the global selector, default selectors, and all user-defined selectors. This property is a read-only property.

You can use the name of a selector as the argument to the StyleManager's `getStyleDeclaration()` method.

The following example stores the names of all the selectors that are registered with the StyleManager in an Array. It then iterates over that Array and displays values for another Array of style properties by passing the selector name to the `getStyleDeclaration()` method.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/SelectorsTest.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        .unusedStyleTest {
            fontSize:17;
            color:green;
        }
    </fx:Style>
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

    <fx:Script>
        <![CDATA[
            private var stylesList:Array = [
                'fontSize', 'color', 'fontWeight', 'fontFamily', 'fontStyle'
            ];

            public function showSelectors():void {
                msg.text = "List all selectors, and show when they explicitly define the
following:\n";
                msg.text += stylesList.toString();
                var selectors:Array = styleManager.selectors;
                for (var i:int = 0; i < selectors.length; i++) {
                    msg.text += "\n\n" + selectors[i] + " {"
                    for (var j:int = 0; j < stylesList.length; j++) {
                        var s:String =
CSSStyleDeclaration(styleManager.getStyleDeclaration(selectors[i])).getStyle(stylesList[j]);
                        if (s != null) {
                            msg.text += "\n    " + stylesList[j] + ":" + s + ";";
                        }
                    }
                    msg.text += "\n}";
                }
            }
        ]]>
    </fx:Script>
    <s:Button label="Show Selectors" click="showSelectors()"/>
    <s:TextArea id="msg" width="100%" height="100%"/>
</s:Application>
```

Creating style declarations with the StyleManager

You can create CSS style declarations by using ActionScript with the CSSStyleDeclaration class. This lets you create and edit style sheets at run time and apply them to classes in your applications. To change the definition of the styles or to apply them during run time, you use the setStyle() method.

The following example creates a new CSSStyleDeclaration object for the Spark and MX Button controls:

```
<?xml version="1.0"?>
<!-- styles/StyleDeclarationTypeSelector.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>
  <fx:Script><![CDATA[
     private var mySparkDynStyle:CSSStyleDeclaration;
     private var myHaloDynStyle:CSSStyleDeclaration;
     private function initApp():void {
        /* These CSSStyleDeclaration objects replace
           all style properties for their types, causing potentially unwanted
           results. */
        var mySparkDynStyle:CSSStyleDeclaration  = new CSSStyleDeclaration();
        var myMXDynStyle:CSSStyleDeclaration  = new CSSStyleDeclaration();
        myMXDynStyle.setStyle('color', 'blue');
        myMXDynStyle.setStyle('fontFamily', 'georgia');
        myMXDynStyle.setStyle('fontSize', 24);
        mySparkDynStyle.setStyle('color', 'blue');
        mySparkDynStyle.setStyle('fontFamily', 'georgia');
        mySparkDynStyle.setStyle('fontSize', 24);
        styleManager.setStyleDeclaration("mx.controls.Button", myMXDynStyle, true);
        styleManager.setStyleDeclaration("spark.components.Button", mySparkDynStyle, true);
    }
  ]]></fx:Script>
  <s:Button id="mySparkButton" label="Spark Button"/>
  <mx:Button id="myHaloButton" label="MX Button"/>
</s:Application>
```

When you set a new CSSStyleDeclaration on a type selector, you are replacing the entire existing type selector with your own selector. All style properties that you do not explicitly define in the new CSSStyleDeclaration are set to `null`. This can remove skins, borders, padding, and other properties that are defined in the default style sheet (defaults.css in the frameworks.swc file), default theme (defaults.css in the spark.swc file) or other style sheet that you may have applied already.

Using the setStyle() and getStyle() methods

You cannot get or set style properties directly on a component as you can with other properties. Instead, you set style properties at run time by using the getStyle() and setStyle() ActionScript methods. When you use the `getStyle()` and `setStyle()` methods, you can access the style properties of instances of objects or of style sheets.

Every component exposes these methods. When you are instantiating an object and setting the styles for the first time, you should try to apply style sheets rather than use the `setStyle()` method because it is computationally expensive. This method should be used only when you are changing an object’s styles during run time. For more information, see “Improving performance with the setStyle() method” on page 1545.

Setting styles

You use the `getStyle()` method in the following way:

```
var:return_type componentInstance.getStyle(property_name)
```

The *return_type* depends on the style that you access. Styles can be of type String, Number, Boolean, or, in the case of skins, Class. The *property_name* is a String that indicates the name of the style property—for example, `fontSize`.

The `setStyle()` method is used like this:

```
componentInstance.setStyle(property_name, property_value)
```

The *property_value* sets the new value of the specified property. To determine valid values for properties, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The following example uses the `getStyle()` and `setStyle()` methods to change the Button's `fontSize` style and display the new size in the TextInput:

```
<?xml version="1.0"?>
<!-- styles/SetSizeGetSize.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">

  <fx:Script><![CDATA[
    import mx.controls.Alert;

    [Bindable]
    private var curSize:int = 10;

    private function initApp():void {
        ip1.setStyle("fontSize", curSize);
        b1.setStyle("fontSize", curSize);
        b2.setStyle("fontSize", curSize);
    }

    public function showStyles():void {
        Alert.show("Font size is " + ip1.getStyle("fontSize") + ".");
     }

     public function setNewStyles():void {
        curSize = Number(ip2.text);

        ip1.setStyle("fontSize", curSize);
        b1.setStyle("fontSize", curSize);
        b2.setStyle("fontSize", curSize);
```

```
      }
   ]]></fx:Script>
   <s:VGroup id="vb">
      <s:TextInput id="ip1"
         styleName="myClass"
         text="This is a TextInput control."
         width="400"
      />
      <s:Label id="lb1" text="Current size: {curSize}" width="400"/>
      <s:Button id="b1" label="Get Style" click="showStyles();"/>
      <s:Form>
         <s:FormItem label="Enter new size:">
             <s:HGroup>
                 <s:TextInput text="{curSize}" id="ip2" width="50"/>
                 <s:Button id="b2" label="Set Style" click="setNewStyles();"/>
             </s:HGroup>
         </s:FormItem>
      </s:Form>
   </s:VGroup>
</s:Application>
```

You can use the `getStyle()` method to access style properties regardless of how they were set. If you defined a style property as a tag property inline rather than in an `<fx:Style>` tag, you can get and set this style. You can override style properties that were applied in any way, such as in an `<fx:Style>` tag or in an external style sheet.

The following example sets a style property inline, and then reads that property with the `getStyle()` method:

```
<?xml version="1.0"?>
<!-- styles/GetStyleInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script><![CDATA[
        private function readStyle():void {
            myLabel.text = "Label style is: " + myLabel.getStyle("fontStyle");
        }
    ]]></fx:Script>

    <s:VGroup width="500" height="200">
        <s:Button id="b1" click="readStyle()" label="Get Style"/>
        <s:Label id="myLabel" fontStyle="italic"/>
    </s:VGroup>
</s:Application>
```

When setting color style properties with the `setStyle()` method, you can use the hexadecimal format or the VGA color name, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatStyleManager.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
  <fx:Script><![CDATA[
     public function initApp():void {
        styleManager.getStyleDeclaration("spark.components.Button").setStyle("color","Blue");
     }
     public function changeStyles(e:Event):void {
        // Check against "255" here, because that is the numeric value of "Blue".
        if (e.currentTarget.getStyle("color") == 255) {
            e.currentTarget.setStyle("color", "Red");
        } else {
            e.currentTarget.setStyle("color", "Blue");
        }
     }
  ]]></fx:Script>
  <s:Button id="myButton" label="Click Here"
        click="changeStyles(event)"/>
</s:Application>
```

When you get a `color` style property with the `getStyle()` method, Flex returns an integer that represents the hexadecimal value of the style property. To convert this to its hexadecimal format, you use the color variable's `toString()` method and pass it the value 16 for the radix (or base):

```
<?xml version="1.0"?>
<!-- styles/ColorFormatNumericValue.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
  <fx:Style>
     @namespace s "library://ns.adobe.com/flex/spark";
     s|Button {
        color: #66CCFF;
     }
  </fx:Style>
  <fx:Script><![CDATA[
     [Bindable]
     private var n:Number;
     private function initApp():void {
        n = myButton.getStyle("color");
     }
     public function changeStyles(e:Event):void {
        if (myButton.getStyle("color").toString(16) == "ff0000") {
           myButton.setStyle("color", 0x66CCFF);
        } else {
           myButton.setStyle("color", "Red");
        }

        n = myButton.getStyle("color"); // Returns 16711680
     }
  ]]></fx:Script>
    <s:VGroup>
        <s:Button id="myButton" label="Click Me" click="changeStyles(event)"/>
        <s:Label id="myLabel" text="0x{n.toString(16).toUpperCase()}"/>
    </s:VGroup>

</s:Application>
```

When you use the `setStyle()` method to change an existing style (for example, to set the `color` property of a Button control to something other than 0x000000, the default), Flex does not overwrite the original style setting. You can return to the original setting by setting the style property to `null`. The following example toggles the color of the Button control between blue and the default by using this technique:

```
<?xml version="1.0"?>
<!-- styles/ResetStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>
    <fx:Script>
        <![CDATA[
        public function toggleStyle():void {
            if (cb1.selected == true) {
                b1.setStyle("color","blue");
                b1.setStyle("fontSize", 8);
            } else {
                b1.setStyle("color", null);
                b1.setStyle("fontSize", null);
            }
        }
        ]]>
    </fx:Script>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Button {
            color: red;
            fontSize: 25;
        }
    </fx:Style>
    <s:Button id="b1" label="Click Me"/>
    <s:CheckBox id="cb1" label="Set Style/Unset Style"
        click="toggleStyle()" selected="false" color="Black"/>
</s:Application>
```

Improving performance with the setStyle() method

Run-time cascading styles are very powerful, but you should use them sparingly and in the correct context. Dynamically setting styles on an instance of an object means accessing the UIComponent’s `setStyle()` method. The `setStyle()` method is one of the most resource-intensive calls in the Flex framework because the call requires notifying all the children of the newly styled object to do another style lookup. The resulting tree of children that must be notified can be quite large.

A common mistake that impacts performance is overusing or unnecessarily using the `setStyle()` method. In general, you need the `setStyle()` method only when you want to change styles on existing objects. Do not use it when setting up styles for an object for the first time. Instead, set styles in an `<fx:Style>` block, through an external CSS style sheet, or as global styles. It is important to initialize your objects with the correct style information if you do not expect these styles to change while your program executes (whether it is your application, a new view in a navigator container, or a dynamically created component).

Some applications must call the `setStyle()` method during the application or object instantiation. If this is the case, call the `setStyle()` method early in the instantiation phase. Early in the instantiation phase means setting styles from the component or application's `preinitialize` event, instead of the `creationComplete` or other event. By setting the styles as early as possible during initialization, you avoid unnecessary style notification and lookup.

For more information about the component startup life cycle, see “Improving startup performance” on page 2333.

Using inline styles

You can set style properties as properties of the component in the MXML tag. Inline style definitions take precedence over any other style definitions. The following example defines a type selector for Button components, but then overrides the `color` with an inline definition:

```
<?xml version="1.0"?>
<!-- styles/InlineOverride.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

  <fx:Style>
     @namespace s "library://ns.adobe.com/flex/spark";
     s|Button {
        fontSize: 10pt;
        fontStyle: italic;
        color: #FF0000;
     }
  </fx:Style>

  <s:Button label="Button Type Selector Color"/>
  <s:Button color="0x999942" label="Inline Color"/>
</s:Application>
```

When setting style properties inline, you must adhere to the ActionScript style property naming syntax rather than the CSS naming syntax. For example, you can set a Button control's `fontSize` property as `font-size` or `fontSize` in an `<fx:Style>` declaration, but you must set it as `fontSize` in a tag definition:

```
<?xml version="1.0"?>
<!-- styles/CamelCase.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

    <fx:Style>
        .myFontStyle {
            fontSize: 15; /* Note the camelCase. */
        }
        .myOtherFontStyle {
            font-size: 15; /* Note the hyphen. */
        }
    </fx:Style>
    <s:Button id="myButton" styleName="myFontStyle" label="Click Me"/>
    <s:Button id="myButton2" styleName="myOtherFontStyle" label="Click Me"/>
</s:Application>
```

When setting color style properties inline, you can use the hexadecimal format or the VGA color name, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>
    <s:Button id="myButton" color="Blue" label="Click Here"/>
    <s:Button id="myButton2" color="0x6666CC" label="Click Here"/>
</s:Application>
```

You can remove an inline style definition by using the `clearStyle()` method.

You can bind inline style properties to variables, as long as you tag the variable as `[Bindable]`. The following example binds the value of the `backgroundColor` property of the HBox controls to the value of the `colorValue` variable:

```
<?xml version="1.0"?>
<!-- styles/PropertyBinding.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

      <fx:Script><![CDATA[
         [Bindable]
         public var colorValue:int = 0x333999;
         public function changeHBoxStyle():void {
            colorValue = cp.selectedColor;
         }
      ]]></fx:Script>
      <mx:HBox width="100" height="100" backgroundColor="{colorValue}"/>
      <mx:ColorPicker id="cp" showTextField="true" change="changeHBoxStyle()"
selectedColor="0x333999"/>
</s:Application>
```

Binding a style property can be a computationally expensive operation. You should use this method of applying style properties only when absolutely necessary. You can also use the `getStyle()` method in a data binding expression.

Loading style sheets at run time

You can load style sheets at run time by using the StyleManager. You can access the top-level StyleManager by using the `styleManager` property of the Application object. These style sheets take the form of SWF files that are dynamically loaded while your application runs.

By loading style sheets at run time, you can load images (for graphical skins), fonts, type and class selectors, and programmatic skins into your application without embedding them at compile time. This lets skins and fonts be partitioned into separate SWF files, away from the main application. As a result, the application's SWF file size is smaller, which reduces the initial download time.

However, the first time a run-time style sheet is used, it takes longer for the styles and skins to be applied than if you load styles by using the `<fx:Style>` tag or set the styles inline. This is because Flex must download the necessary CSS-based SWF file while the application is starting up or running.

Loading style sheets at run time is a three-step process:

1 Write a CSS file for your application.

2 Compile the CSS file into a SWF file.

3 Call the `styleManager.loadStyleDeclarations()` method in your application. This method loads the CSS-based SWF file into your application. When this method executes, Flex loads the new CSSStyleDeclarations into the top-level StyleManager.

You can load multiple style sheets that define the same styles in the same application. After you set a style, subsequent style sheets can overwrite previous ones if they have common selectors. Styles loaded with run-time style sheets do not completely replace compile-time styles, however. They just override them until the run-time style sheets are unloaded. At that point, Flex reverts to the compile-time style settings. Compile-time style settings include any default styles sheets that were loaded at compile time, theme files loaded by using the `theme` compiler option, and styles set by using the `<fx:Style>` block inside an MXML file.

You cannot load an uncompiled CSS file into your application at run time. You must compile it into a SWF file before loading it.

Creating a run-time style sheet

To load style sheets at run time, you must first create a style sheet that is compiled into a SWF file. A run-time style sheet is like any other style sheet. It can be a simple style sheet that sets basic style properties, as the following example shows:

```
/* ../assets/BasicStyles.css */
@namespace s "library://ns.adobe.com/flex/spark";
s|Button {
    fontSize:    24;
    color: #FF9933;
}
s|Label {
    fontSize:    24;
    color: #FF9933;
}
```

Or the style sheet can be a complex style sheet that embeds programmatic and graphical skins, fonts, and other style properties, and uses type and class selectors, as the following example shows:

```
/* assets/ComplexStyles.css */
@namespace mx "library://ns.adobe.com/flex/mx";
mx|Application {
    backgroundImage: "greenBackground.gif";
    theme-color: #9DBAEB;
}
mx|Button {
    fontFamily: Tahoma;
    color: #000000;
    fontSize: 11;
    fontWeight: normal;
    text-roll-over-color: #000000;
    upSkin: Embed(source="orb_up_skin.gif");
    overSkin: Embed(source="orb_over_skin.gif");
    downSkin: Embed(source="orb_down_skin.gif");
}
.noMargins {
    margin-right: 0;
    margin-left: 0;
    margin-top: 0;
    margin-bottom: 0;
    horizontal-gap: 0;
    vertical-gap: 0;
}
```

To create a new style sheet in Flash Builder, you select File > New > CSS File. Create the CSS file in the project's main directory or another subdirectory that is not the bin directory. You should not create the CSS file in the bin directory. Flash Builder will compile the SWF file to the bin directory for you.

Compiling the CSS-based SWF file

Before you can load a style sheet at run time, you must compile the style sheet into a SWF file. The style sheet that you compile into a SWF file must use a .css filename extension.

To compile the CSS file into a SWF file, you use the mxmlc command-line compiler or Flash Builder's compiler. The default result of the compilation is a SWF file with the same name as the CSS file, but with the .swf extension.

The following example produces the BasicStyles.swf file by using the mxmlc command-line compiler:

```
mxmlc BasicStyles.css
```

To compile the SWF file with Flash Builder, right-click the CSS file and select Compile CSS to SWF. Flash Builder saves the SWF file in the project's bin directory. If the original CSS file is in the bin directory, you cannot compile it into a SWF file. You must move it to a different directory before you can compile it.

When you compile your application, the compiler does not perform any compile-time link checking against the CSS-based SWF files used by the application. This means that you are not required to create the SWF file before you compile your main application. This also means that if you mistype the name or location of the SWF file, or if the SWF file does not exist, the application will fail silently. The application will not throw an error at run time.

Loading the style sheets

You load a CSS-based SWF file at run time by using the StyleManager's `loadStyleDeclarations()` method. You can access the top-level StyleManager by using the `styleManager` property of the Application object.

The following example shows loading a style sheet SWF file:

```
styleManager.loadStyleDeclarations("../assets/MyStyles.swf");
```

The first parameter of the `loadStyleDeclarations()` method is the location of the style sheet SWF file to load. The location can be local or remote.

The second parameter is `update`. You set this to `true` or `false`, depending on whether you want the style sheets to immediately update in the application. For more information, see "Updating CSS-based SWF files" on page 1551.

The next parameter, `trustContent`, is optional and obsolete. If you do specify a value, set this to `false`.

The final two parameters are `applicationDomain` and `securityDomain`. These parameters specify the domains into which the style sheet SWF file is loaded. In most cases, you should accept the default values (`null`) for these parameters. The result is that the style sheet's SWF file is loaded into child domains of the current domains. For information on when you might use something other than the default for these parameters, see "Using run-time style sheets with modules and sub-applications" on page 1554.

The following example loads a style sheet when you click the button:

```
<?xml version="1.0"?>
<!-- styles/BasicApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
        public function applyRuntimeStyleSheet():void {
            styleManager.loadStyleDeclarations("assets/BasicStyles.swf")
        }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label text="Click the button to load a new CSS-based SWF file."/>
        <s:Button id="b1" label="Click Me" click="applyRuntimeStyleSheet()"/>
    </s:VGroup>

</s:Application>
```

Loading a remote style sheet typically requires a crossdomain.xml file that gives the loading application permission to load the SWF file. You can do without a crossdomain.xml file if your application is in the local-trusted sandbox, but this is usually restricted to SWF files that have been installed as applications on the local machine.

For more information about crossdomain.xml files, see "Using cross-domain policy files" on page 125.

Also, to use remote style sheets, you must compile the loading application with network access (have the `use-network` compiler property set to `true`, the default). If you compile and run the application on a local file system, you might not be able to load a remotely accessible SWF file.

The `loadStyleDeclarations()` method is asynchronous. It returns an instance of the IEventDispatcher class. You can use this object to trigger events based on the success of the style sheet's loading. You have access to the `StyleEvent.PROGRESS`, `StyleEvent.COMPLETE`, and `StyleEvent.ERROR` events of the loading process.

The following application calls a method when the style sheet finishes loading:

```
<?xml version="1.0"?>
<!-- styles/StylesEventApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init()">

    <fx:Script>
        <![CDATA[
        import mx.events.StyleEvent;

        public function init():void {
            var myEvent:IEventDispatcher =
styleManager.loadStyleDeclarations("../assets/ACBStyles.swf");
            myEvent.addEventListener(StyleEvent.COMPLETE, getImage);
        }
        private function getImage(event:StyleEvent):void {
            map1.source = acb.getStyle("dottedMap");
        }
        ]]>
    </fx:Script>

    <mx:ApplicationControlBar id="acb" width="100%">
        <s:Image id="map1"/>
    </mx:ApplicationControlBar>

</s:Application>
```

The style sheet used in this example embeds a PNG file:

```
/* assets/ACBStyles.css */
@namespace mx "library://ns.adobe.com/flex/mx";
mx|ApplicationControlBar {
    borderStyle:      "solid";
    cornerRadius:     10;
    backgroundColor: #FF9933;
    alpha:            1;
    dottedMap:        "beige_dotted_map.png";
}
```

Updating CSS-based SWF files

You can force an immediate update of all styles in the application when you load a new CSS-based SWF file. You can also delay the update if you want.

The second parameter of the `loadStyleDeclarations()` method is `update`. Set the `update` parameter to `true` to force an immediate update of the styles. Set it to `false` to avoid an immediate update of the styles in the application. The styles are updated the next time you call this method or the `unloadStyleDeclarations()` method with the `update` property set to `true`.

Each time you call the `loadStyleDeclarations()` method with the `update` parameter set to `true`, Adobe® Flash Player and Adobe AIR™ reapply all styles to the display list, which can degrade performance. If you load multiple CSS-based SWF files at the same time, you should set the `update` parameter to `false` for all but the last call to this method. As a result, Flash Player and AIR apply the styles only once for all new style SWF files rather than once for each new style SWF.

The following example loads three style SWF files, but does not apply them until the third one is loaded:

```
<?xml version="1.0"?>
<!-- styles/DelayUpdates.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init()">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

    <fx:Script>
        <![CDATA[
        import mx.events.StyleEvent;

        public function init():void {
            styleManager.loadStyleDeclarations("assets/ButtonStyles.swf", false);
            var myEvent:IEventDispatcher =
              styleManager.loadStyleDeclarations("assets/LabelStyles.swf", false);
            myEvent.addEventListener(StyleEvent.COMPLETE, doUpdate);
        }
        public function doUpdate(event:StyleEvent):void {
            styleManager.loadStyleDeclarations("assets/ACBStyles.swf", true);
        }
        ]]>
    </fx:Script>

    <s:Label text="This is a Label control."/>
    <mx:ApplicationControlBar id="acb" width="100%">
        <s:Button label="Submit"/>
    </mx:ApplicationControlBar>

</s:Application>
```

Unloading style sheets at run time

You can unload a style sheet that you loaded at run time. You do this by using the StyleManager’s `unloadStyleDeclarations()` method. You can access the top-level StyleManager by using the `styleManager` property of the Application object. The result of this method is that all style properties set by the specified style SWF files are returned to their defaults.

The following example loads and unloads a style SWF when you toggle the check box:

```
<?xml version="1.0"?>
<!-- styles/UnloadStyleSheets.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
     <s:VerticalLayout/>
  </s:layout>
    <fx:Script>
        <![CDATA[
        public function toggleStyleSheet():void {
            if (cb1.selected == true) {
                styleManager.loadStyleDeclarations("assets/ButtonStyles.swf", true);
                styleManager.loadStyleDeclarations("assets/LabelStyles.swf", true);
            } else {
                styleManager.unloadStyleDeclarations("assets/ButtonStyles.swf", true);
                styleManager.unloadStyleDeclarations("assets/LabelStyles.swf", true);
            }
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Submit"/>

    <s:Label id="l1" text="This is a Label control."/>

    <s:CheckBox id="cb1" label="Load style sheet"
        click="toggleStyleSheet()" selected="false"/>
</s:Application>
```

The `unloadStyleDeclarations()` method takes a second parameter, `update`. As with loading style sheets, Flash Player and AIR do not reapply the styles (in this case, reapply default styles when the loaded styles are unloaded) if you set the value of the `update` parameter to `false`. For more information about the `update` parameter, see "Loading the style sheets" on page 1549.

Using run-time style sheets in custom components

You can use run-time style sheets in custom components. To do this, you generally call the `loadStyleDeclaration()` method after the component is initialized. If the style sheet contains class selectors, you then apply them by setting the `styleName` property.

The following example defines style properties and skins in a class selector named `specialStyle`:

```
/* ../assets/CustomComponentStyles.css */
.specialStyle {
    fontSize:    24;
    color: #FF9933;
    upSkin: Embed(source="SubmitButtonSkins.swf", symbol="MyUpSkin");
    overSkin: Embed(source="SubmitButtonSkins.swf", symbol="MyOverSkin");
    downSkin: Embed(source="SubmitButtonSkins.swf", symbol="MyDownSkin");
}
```

The following example custom component loads this style sheet and applies the `specialStyle` class selector to itself during initialization:

```
// styles/MyButton.as -->
package {
    import mx.controls.Button;
    import mx.events.*;

    public class MyButton extends Button {

        public function MyButton() {
            addEventListener(FlexEvent.INITIALIZE, initializeHandler);
        }
         // Gets called when the component has been initialized
        private function initializeHandler(event:FlexEvent):void {
            styleManager.loadStyleDeclarations("assets/CustomComponentStyles.swf");
            this.styleName = "specialStyle";
        }
    }
}
```

The following sample application uses this custom button:

```
<?xml version="1.0"?>
<!-- styles/MyButtonApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:custom="*">

    <custom:MyButton/>

</s:Application>
```

Using theme SWC files as run-time style sheets

If you have an existing theme SWC file, you can use it as a run-time style sheet. To do this, you must extract the CSS file from the theme SWC file. You then compile the style SWF file by passing the remaining SWC file as a library.

The following steps show this process using the command line:

1 Extract the CSS file from the SWC file using PKZip or similar archiving utility, as the following example shows:

```
$ unzip halo.swc defaults.css
```

2 (Optional) Rename the CSS file to a meaningful name. This is the name of your style SWF file. The following example renames the defaults.css file to halo.css:

```
$ mv defaults.css halo.css
```

3 Compile the style SWF file. Add the theme SWC file to the style SWF file by using the `include-libraries` option, as the following example shows:

```
$ mxmlc -include-libraries=halo.swc halo.css
```

If you have multiple CSS files inside a theme SWC file, you must extract all of them before compiling the style SWF file.

Using run-time style sheets with modules and sub-applications

When loading run-time style sheets with modules and sub-applications, consider the following factors:

- The application domain into which you load the style SWF file

- The StyleManager that loads the style SWF file

Selecting the application domain that you load the style SWF file into determines where the classes are loaded. You typically load a style SWF file into the module's *current application domain*. You do not typically load it into a child application domain of the current module or application.

The StyleManager that loads the style SWF file is equally important. Because each module and application has its own StyleManager, you can load a style SWF file with the root application's StyleManager or any module's StyleManager. Which StyleManager you choose determines which applications or modules in the chain use those styles.

To load the style sheet into the current application domain of the module, set the *`applicationDomain`* parameter to `ApplicationDomain.currentDomain` in the `loadStyleDeclarations()` method. The following example loads the Style.swf file in the module into the current application domain:

```
styleManager.loadStyleDeclaration("Style.swf",true,false, ApplicationDomain.currentDomain)
```

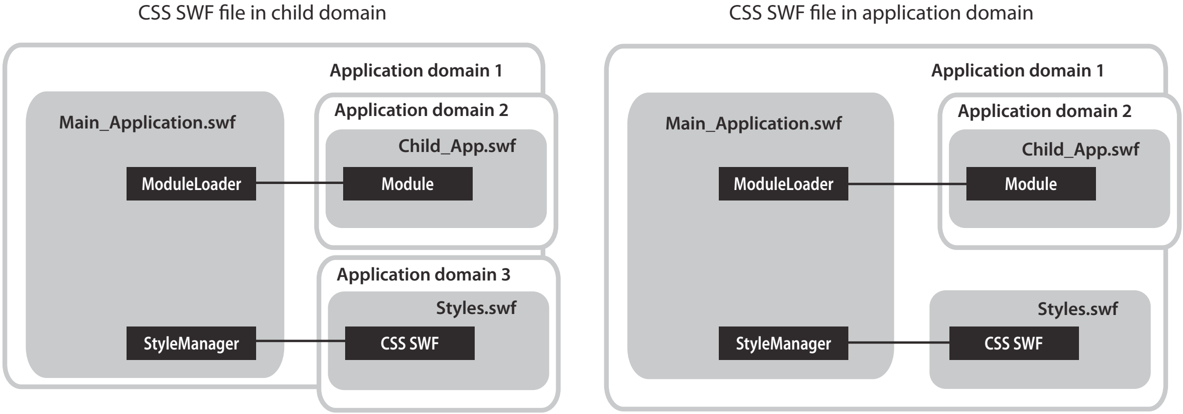
If you leave the *`applicationDomain`* parameter blank, then the default is to load the SWF file into a child domain of the main application. The result is a style SWF that is in a sibling application domain of modules and sub-applications.

Each module and the main application have their own instances of type IStyleManager2. If you want a style SWF file to be used by all modules, load it with the main application's StyleManager. If you want a style SWF file to be used by a module and its children, load the style SWF file with the module's StyleManager.

If you load the style SWF file into the main application's StyleManager, then all child modules and sub-applications will merge their styles with it when they are loaded. If you load the style SWF file into a module or sub-application's StyleManager, then only the children of that module or sub-application will merge their styles with it.

To load a style SWF file with the current module or application's StyleManager, use the `styleManager` property. In general, you should not load a style SWF file into a StyleManager of another module or application from one module or application.

Even when loading a style SWF file into the main application, you should specify the current application domain rather than a child application domain to avoid class conflicts. The following example illustrates loading the run-time style sheet into a child application domain and the current application domain of the main application:



In the first approach, the module and the run-time style sheet are loaded into separate child application domains (application domains 2 and 3). Because the style SWF file and the module are in sibling application domains, the classes referenced in the style module (such as skins) might cause class conflicts with the classes in the module.

The second approach loads the module into a child application domain and the style SWF file into the same domain as the main application. In this case, no class conflicts should occur.

The disadvantage of loading a style SWF file into the current application domain is that you cannot unload it, even when the module is unloaded.

More Help topics

"Using styles with modules" on page 143

Using filters

You can use filters to apply style-like effects to MX and Spark components. You can apply filters to any visual component that is derived from UIComponent. Filters are not styles; you cannot apply them with a style sheet or the `setStyle()` method. The result of a filter, such as a drop shadow, is often thought of as a style.

Filters are in the spark.filters.* package, and include the DropShadowFilter, GlowFilter, and BlurFilter classes. There are equivalent filter classes in the mx.filters.* package. These filters are mapped to the spark.filters.* package by the compiler.

To apply a filter to a component with MXML, you add the filter class to the component's `filters` Array. The `filters` Array contains any number of filters you want to apply to the component.

You wrap the filter class in an `<mx:filters>` or `<s:filters>` tag to define the array of filters applied to that control. If the control is an MX control, you use the `<mx:filters>` tag. If the control is a Spark control, you use the `<s:filters>` tag.

The following example applies a drop shadow to Label controls by using expanded MXML syntax and inline syntax:

```
<?xml version="1.0"?>
<!-- styles/ApplyFilterInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:VGroup>
        <!-- Apply filter using MXML syntax to set properties. -->
        <s:Label text="DropShadowFilter" fontSize="20">
            <s:filters>
                <s:DropShadowFilter distance="10" angle="45"/>
            </s:filters>
        </s:Label>
        <!-- Apply filter and set properties inline. -->
        <s:Label text="DropShadowFilter (inline)"
            fontSize="20"
            filters="{[new DropShadowFilter(10, 45)]}"/>
    </s:VGroup>
</s:Application>
```

You can apply filters in ActionScript. You do this by importing the spark.filters.* package, and then adding the new filter to the `filters` Array of the Flex control. The following example toggles a shadow filter on the Label control when the user clicks the button:

```
<?xml version="1.0"?>
<!-- styles/ApplyFilterAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script><![CDATA[
        import spark.filters.*;
        private var dsf:DropShadowFilter;
        public function toggleFilter():void {
            trace(label1.filters.length);
            if (label1.filters.length == 0) {
                /*
                    The first four properties of the DropShadowFilter constructor are
                    distance, angle, color, and alpha.
                */
                dsf = new DropShadowFilter(5,30,0x000000,.8);
                label1.filters = [dsf];
            } else {
                label1.filters = null;
            }
        }
    ]]></fx:Script>

    <s:VGroup>
        <s:Label id="label1" text="ActionScript-applied filter."/>
        <s:Button id="b1" label="Toggle Filter" click="toggleFilter()"/>
    </s:VGroup>
</s:Application>
```

You cannot bind the filter properties to other values.

If you change a filter, you must reassign it to the component so that the changes take effect. The following example changes the color of the filters when you click the button:

```
<?xml version="1.0"?>
<!-- styles/FilterChange.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="createFilters()"
    height="700">

  <fx:Script><![CDATA[
     import spark.filters.*;
     import flash.filters.BitmapFilterQuality;
     import flash.filters.BitmapFilterType;

     private var myBlurFilter:BlurFilter;
     private var myGlowFilter:GlowFilter;
     private var myBevelFilter:BevelFilter;
     private var myDropShadowFilter:DropShadowFilter;

     private var color:Number = 0xFF33FF;
     public function createFilters():void {

        myBlurFilter = new BlurFilter(4, 4, 1);
        myGlowFilter = new GlowFilter(color, .8, 6, 6, 2, 1,
            false, false);
        myDropShadowFilter = new DropShadowFilter(15, 45,
            color, 0.8, 8, 8, 0.65, 1, false, false);
        myBevelFilter = new BevelFilter(5, 45, color, 0.8, 0x333333,
            0.8, 5, 5, 1, BitmapFilterQuality.HIGH, BitmapFilterType.INNER,
            false);

        applyFilters();
     }

     public function applyFilters():void {
        rte1.filters = [myGlowFilter];
        b1.filters = [myDropShadowFilter];
        dc1.filters = [myBevelFilter];
        hs1.filters = [myBlurFilter];
     }
     public function changeFilters():void {
        color = 0x336633;
        createFilters();
     }
  ]]></fx:Script>
    <s:VGroup>
        <mx:RichTextEditor id="rte1"/>
        <mx:DateChooser id="dc1"/>
        <mx:HSlider id="hs1"/>
        <mx:Button id="b1" label="Click me" click="changeFilters()"/>
    </s:VGroup>
</s:Application>
```

More Help topics

“Filters in FXG and MXML graphics” on page 1778

“Using filters with chart controls” on page 1213

“Using Animated filters” on page 1560

Clearing filters

You can remove filters by setting the filters Array to an empty Array, as the following example shows:

```
<s:filters>
    <fx:Array/>
</s:filters>
```

Using the dropShadowEnabled property

Some MX containers and controls have a `dropShadowEnabled` property. You can use this property instead of applying a filter if you are using the Halo theme. This property is more efficient than using a filter.

To use the `dropShadowEnabled` property, set it to `true` on the component. However, when you use this property, the component draws its own drop shadow by using the RectangularDropShadow class. It copies the edges of the target and then draws the shadow onto a bitmap, which is then attached the target. If the target component is rotated, the drop shadow might appear jagged because of the way rotated vectors are rendered.

To avoid this and have smooth drop shadow filters on rotated components, you use the DropShadowFilter class.

The following example shows the difference between drawing a filter with the `dropShadowEnabled` property and using the DropShadowFilter class. This example was compiled with the Halo theme:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/SmoothFilter.mxml -->
<!-- Compile this example by setting theme=halo.swc for a compiler argument. -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    backgroundColor="0xFFFFFF">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Canvas {
            borderStyle:solid;
            cornerRadius:10;
            borderColor:#000000;
            backgroundColor:#FFFFFF;
        }
    </fx:Style>
    <fx:Script>
        <![CDATA[
            import spark.filters.*;
            import flash.filters.BitmapFilterQuality;
            private function getBitmapFilter():DropShadowFilter {
                var distance:Number = 3;
                var angle:Number = 90;
                var color:Number = 0x000000;
                var alpha:Number = 1;
                var blurX:Number = 8;
                var blurY:Number = 8;
                var strength:Number = 0.65;
                var quality:Number = BitmapFilterQuality.LOW;
                var inner:Boolean = false;
                var knockout:Boolean = false;
                return new DropShadowFilter(distance, angle, color, alpha,
                    blurX, blurY, strength, quality, inner, knockout);
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <!-- This rotated canvas applies a filter using the dropShadowEnabled
            property. As a result, the filter's edges are slightly jagged. -->
        <mx:Canvas id="canvas1"
            dropShadowEnabled="true"
            creationComplete="canvas1.rotation=-10"
            x="50" y="80"
            width="200"
            height="200"/>
        <!-- This rotated canvas applies a bitmap filter. As a result,
            the edges are smoother. -->
        <mx:Canvas id="canvas2"
            filters="{[getBitmapFilter()]}"
            creationComplete="canvas2.rotation=-10"
            x="50" y="450"
            width="200"
            height="200"/>
    </s:HGroup>
</s:Application>
```

Using Animated filters

The spark.effects.AnimateFilter class is a more generic version of what effects such as the BlurFilter and GlowFilter classes do. It lets you apply any filter and optionally animate the filter during the context of a Spark component's effect or transition.

The AnimateFilter class extends the Animate class, and provides an additional input property `bitmapFilter`, of type IBitmapFilter. It lets you animate an arbitrary set of properties of the filter between values, as specified by the `propertyValuesList`.

The AnimateFilter class applies the associated filter when the effect begins, and removes it when the effect finishes.

The primary difference between the Animate class and the AnimateFilter class, is that the properties that the effect is animating apply not to the target of the effect, but to the associated filter instead. As a result, for example, the extents of a drop shadow can be animated during an effect sequence.

The following example defines a BlurFilter and the animation on that filter. It then plays the animation when the user moves the mouse pointer over the Button control, and stops the animation when the user moves the mouse away from the Button control.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/AnimateFilterExample.mxml -->
<s:Application backgroundColor="0xFFFFFF"
     xmlns:fx="http://ns.adobe.com/mxml/2009"
     xmlns:mx="library://ns.adobe.com/flex/mx"
     xmlns:s="library://ns.adobe.com/flex/spark"
     creationComplete="initApp();">
     <fx:Script>
          <![CDATA[
               import spark.effects.*;
               import spark.filters.BlurFilter;
               import spark.effects.animation.*;

               private var blurFilter:BlurFilter;
               private var blurAnim:AnimateFilter;
               private var smpX:SimpleMotionPath;
               private var smpY:SimpleMotionPath;

               private function initApp():void {
                    blurFilter = new BlurFilter();
                    blurAnim = new AnimateFilter(btn1, blurFilter);
                    smpX = new SimpleMotionPath("blurX",0,20);
                    smpY = new SimpleMotionPath("blurY",0,20);
                    blurAnim.motionPaths = Vector.<MotionPath>([smpX,smpY]);
               }

               private function doBlurSample():void {
                    blurAnim.repeatCount = 0;
                    blurAnim.repeatBehavior = RepeatBehavior.REVERSE;
                    blurAnim.play();
               }

               private function stopAnimation():void {
                    blurAnim.stop();
               }
          ]]>
     </fx:Script>
      <s:Button id="btn1" label="Blur"
           mouseOver="doBlurSample();"
           mouseOut="stopAnimation();"/>
</s:Application>
```

About themes

A *theme* defines the look and feel of an application's visual components. A theme can define something as simple as the color scheme or common font for an application, or it can be a complete reskinning of all the components used by the application.

The default theme for Flex 4 components is Spark. All controls in an application built with Flex 4 use the Spark theme. Even MX controls have a Spark look and feel in a Flex 4 application.

The Spark theme is a combination of styles and skin classes that define the appearance of the components in the spark.components package. This theme is defined in the defaults.css style sheet in the spark.swc file. Spark components use the skin classes in the spark.skins.spark.* package. MX components use the skin classes in the mx.skins.spark.* package.

For Flex 3, the default theme was called Halo. The components that used the Halo theme are in the mx.controls.* package. They are typically referred to as MX components. In Flex 4, however, the MX components use the Spark theme to define their appearance by default. This is set in the defaults.css file in the framework.swc file.

To use the Halo theme in a Flex 4 application, you can use the `theme` compiler option to point to the Halo theme SWC file, or you can set the `compatibility-version` compiler option to 3.0.0. If you use the Halo theme, then the Halo theme is applied only to MX components in your application. The Spark components continue to use the Spark theme unless you specifically override them.

Themes usually take the form of a SWC file. However, themes can also be a CSS file and embedded graphical resources, such as symbols from a SWF file. Theme SWC files can also be compiled into style SWF files so that they can be loaded at run time. For more information, see "Using theme SWC files as run-time style sheets" on page 1554.

To apply a theme SWC file to your application, use the instructions in "Using themes" on page 1562. To create your own theme, use the instructions in "Creating a theme SWC file" on page 1565.

Flex also includes several other predefined themes that you can apply to your applications. For more information, see "About the included theme files" on page 1565.

Using themes

Themes generally take the form of a theme SWC file. These SWC files contain style sheets and skinning assets. You use the assets inside theme SWC files for programmatic skins or graphical assets, such as SWF, GIF, or JPEG files. Themes can also contain just stand-alone CSS files.

Packaging a theme as a SWC file rather than as a loose collection of files has the following benefits:

- SWC files are easier to distribute.
- SWC files cannot be modified and reapplied to the application without recompiling.
- SWC files are precompiled. This reduces application compile time, compared to compile time for skin classes that are not in a SWC file.

You apply a theme to your application by specifying the SWC or CSS file with the `theme` compiler option. The following example uses the mxmlc command-line compiler to compile an application that uses the BullyBuster theme SWC file:

```
mxmlc -theme theme/BullyBuster.swc MainApp.mxml
```

When compiling an application by using options in the flex-config.xml file, you specify the theme as follows:

```
<compiler>
    <theme>
        <filename>c:/theme/BullyBuster.swc</filename>
    </theme>
</compiler>
```

When you add a SWC file to the list of themes, the compiler adds the classes in the SWC file to the application's `library-path`, and applies any CSS files contained in the SWC file to your application. The converse is not true, however. If you add a SWC file to the `library-path`, but do not specify that SWC file with the `theme` option, the compiler does not apply CSS files in that SWC file to the application.

For more information, see "Flex compilers" on page 2164.

Themes are additive. You can specify more than one theme file to be applied to the application. If there are no overlapping styles, both themes are applied completely. The ordering of the theme files is important, though. If you specify the same style property in more than one theme file, Flex uses the property from the last theme in the list. In the following example, style properties in the Wooden.css file take precedence, but unique properties in the first two theme files are also applied:

```
<compiler>
    <theme>
        <filename>../themes/Institutional.css</filename>
        <filename>../themes/Ice.css</filename>
        <filename>../themes/Wooden.css</filename>
    </theme>
</compiler>
```

On the command line, you can use the `+=` operator with the `theme` compiler option to apply themes additively. The following example applies the BullyBuster.swc theme, in addition to the default theme:

```
mxmlc -theme+=themes/BullyBuster.swc MainApp.mxml
```

Creating color schemes

One common way to use themes is to create a color scheme for your application. This color scheme can be used to blend your application into its surrounding website, or define its overall look and feel.

In Spark applications, most styles are inheritable styles. This means that you can set any style on any level of the containment hierarchy. The inheritable styles will be inherited by all controls that are within the container that you set it on. One common use of this is to set styles on the Application tag so that the entire application uses those styles.

The following table describes the color styles that you use most often to create a color scheme in Spark applications:

Property	Description
`backgroundColor`	The background color of the application.
`chromeColor`	The base color of the component. The default skins are based on various shades of gray. The `chromeColor` style is applied as a color transformation (tinting), which preserves all of the visual fidelity of the default appearance. Items colorized by other color style values are not affected by `chromeColor`.
`color`	The color of text in a component.
`contentBackgroundColor`	The color of the content area of components that contain content.
`focusColor`	The color of the focus glow.
`rollOverColor`	The background color of items when the mouse is positioned over the item.
`selectionColor`	The background color of selected items in a drop down list.
`symbolColor`	The color of symbols or glyphs. For example, the arrows on a scroll bar or the check mark in a CheckBox control.

The following example lets you toggle between two color schemes. It shows that the colors are inherited by all Spark controls in that application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/SparkThemeColorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
            import mx.core.FlexGlobals;
            import mx.events.IndexChangedEvent;

            private function changeStyleSettings(e:Event):void {
                if (e.currentTarget.selectedItem == "Gray") {
           FlexGlobals.topLevelApplication.setStyle("backgroundColor", 0x333333);
                    FlexGlobals.topLevelApplication.setStyle("chromeColor", 0x4C4C4C);
                    FlexGlobals.topLevelApplication.setStyle("color", 0xCCCCCC);
                FlexGlobals.topLevelApplication.setStyle("contentBackgroundColor", 0x555555);
                    FlexGlobals.topLevelApplication.setStyle("symbolColor", 0xFFFFFF);
                    FlexGlobals.topLevelApplication.setStyle("rollOverColor", 0x666666);
                    FlexGlobals.topLevelApplication.setStyle("selectionColor", 0x999999);
                    FlexGlobals.topLevelApplication.setStyle("focusColor", 0xEEEEEE);
                } else {
        FlexGlobals.topLevelApplication.setStyle("backgroundColor",0xCCCC99);
                    FlexGlobals.topLevelApplication.setStyle("chromeColor", 0x999966);
                    FlexGlobals.topLevelApplication.setStyle("color", 0x996600);
                FlexGlobals.topLevelApplication.setStyle("contentBackgroundColor", 0xFFFFCC);
                    FlexGlobals.topLevelApplication.setStyle("symbolColor", 0x663300);
                    FlexGlobals.topLevelApplication.setStyle("rollOverColor", 0xFFEE88);
                    FlexGlobals.topLevelApplication.setStyle("selectionColor", 0xFFCC66);
              FlexGlobals.topLevelApplication.setStyle("focusColor", 0xCC9900);
                }
            }
        ]]>
    </fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Button label="Button"/>
    <s:ToggleButton label="Toggle Button"/>
    <s:CheckBox label="CheckBox" selected="true"/>
    <s:RadioButton label="RadioButton" selected="true"/>
    <s:NumericStepper/>
    <s:TextInput/>
    <s:TextArea/>
    <s:DropDownList id="myDropDown" selectedIndex="0"
        change="changeStyleSettings(event);">
        <s:dataProvider>
            <s:ArrayList source="[Gray,Brown]"/>
        </s:dataProvider>
    </s:DropDownList>
</s:Application>
```

You can affect the general coloring of MX components without creating a new theme or changing many of the style properties in the Halo theme. You do this with the `themeColor` property. For more information, see “Creating a Halo theme” on page 1661.

A useful utility for developing color schemes is Kuler. This utility helps users create new color schemes and share them with other users. For more information, see the Kuler online application at http://kuler.adobe.com/.

About the included theme files

Flex includes several themes that you can use. Some are for version compatibility, some are for testing, and some are purely for visual appearance. Most themes are located in the *sdk_install_dir*/frameworks/themes directory for the SDK. For Flash Builder, the themes are located in the *flash_builder_install_dir*/sdks/*sdk_version*/frameworks/themes directory. Other themes are in the /*sdk_version*/samples/themes directory.

The following table describes these themes:

Theme	Description
Aeon Graphical	Contains the AeonGraphical.css file and the AeonGraphical.swf file. This is the graphical version of the Halo theme, which is only used by MX components. The source FLA file for the graphical assets is in the AeonGraphical Source directory. You change the graphical assets in this file by using the Flash IDE. You must ensure that you export the SWF file, which is used by the theme in Flex.
Halo	The Halo theme uses the halo.swc theme file and the classes in the mx.skins.halo package. For more information about the default theme assets, see "About the default style sheets" on page 1534.
Ice	Contains the Ice.css file.
Institutional	Contains the Institutional.css file.
Mobile	Contains the defaults.css file for mobile applications. For more information, see Mobile theme.
Sample themes	The themes in the samples/themes directory include arcade, cobalt, and graphite. These theme files should be considered sample quality. Before you use them in production applications, be sure to test them thoroughly.
Smoke	Contains the Smoke.css file and a background JPEG file.
Wireframe	An alternate theme for the Spark components. This theme gives a simple, wireframe appearance to an application. It might be appropriate to use it when developing a prototype or "proof of concept" application. This theme is also useful as a starting point to write custom skins, particularly if they are not styleable. This is because the skin classes in the Wireframe theme file are simpler than their Spark counterparts.
Wooden	Contains the Wooden.css file and a background JPEG file.

To apply a theme to an application, you use the `theme` compiler option, as the following example shows:

```
mxmlc -theme=c:\Flex\frameworks\themes\Halo\halo.swc MyApp.mxml
```

In addition to the basic themes, Flex also includes the MXFTEText.css theme file. This file is located in the frameworks directory. You use this theme if you want to use embedded fonts with MX controls in your Flex 4 applications. For more information, see "Embedding fonts with MX components" on page 1592.

Another theme file, MXCharts.css, is meant to be used with MX-only applications that contain charting components.

Creating a theme SWC file

To build a theme SWC file, you create a CSS file, and include that file plus the graphical and programmatic assets in the SWC file. To do this, you use the `include-file` option of the compc utility.

A good starting point for creating a new theme is to customize the default CSS files for the Spark and MX components. For Spark components, customize the defaults.css file in the spark.swc file. For MX components, customize the defaults.css file in the framework.swc file. The following sections describe these steps in detail.

Creating a theme style sheet

A theme typically consists of a style sheet and any assets that the style sheet uses. These assets can be graphic files or programmatic skin classes. For the Spark theme, the assets are skin classes that use a combination of MXML and FXG syntax to draw the skin. For the Halo theme, the assets are typically ActionScript class files that extend Halo skin classes.

To add MXML skins to a Spark theme file, you use a `ClassReference` statement to specify the class name (not the file name) for each component. You must also specify a namespace and the Spark skin class for that component. The following is a typical CSS file that defines a small Spark theme:

```
@namespace s "library://ns.adobe.com/flex/spark";
s|Button {
    skinClass: ClassReference("ButtonTransitionSkin");
}
s|CheckBox {
    skinClass: ClassReference("CheckBoxTransitionSkin");
}
```

To add programmatic skin classes to a Halo theme's CSS file, you specify a class for each of the component's states, as the following example shows:

```
@namespace mx "library://ns.adobe.com/flex/mx";
mx|Button {
    upSkin:ClassReference('myskins.ButtonSkin');
    downSkin:ClassReference('myskins.ButtonSkin');
    overSkin:ClassReference('myskins.ButtonSkin');
}
```

For information about creating custom Spark skins, see “Spark Skinning” on page 1602. For information about creating custom Halo skins, see “Creating programmatic skins for MX components” on page 1674.

To add graphic files to the theme's CSS file, which is a common task for the Halo theme, you use the `Embed` statement. The following example defines new graphical skins for the Button class:

```
@namespace mx "library://ns.adobe.com/flex/mx";
mx|Button {
    upSkin: Embed("upIcon.jpg");
    downSkin: Embed("downIcon.jpg");
    overSkin: Embed("overIcon.jpg");
}
```

The name you provide for the `Embed` keyword is the same name that you use for the skin asset when compiling the SWC file.

The CSS file can include any number of class selectors, as well as type selectors. Style definitions for your theme do not have to use skins. You can simply set style properties in the style sheet, as the following example shows:

```
@namespace mx "library://ns.adobe.com/flex/mx";
mx|ControlBar {
    color:red;
}
```

For more information about the default style sheets, see “About the default style sheets” on page 1534.

Compiling a theme SWC file

You compile a theme SWC file by using the `include-file` and `include-classes` options of the component compiler. This is the same compiler that creates component libraries and Runtime Shared Libraries (RSLs).

You invoke the component compiler either with the compc command line utility or when creating a Library Project in Adobe Flash Builder.

You use the `include-file` option to add the CSS file and graphics files to the theme SWC file. You use the `include-classes` option to add programmatic skin classes to the theme SWC file.

To simplify the commands for compiling theme SWC files, you can use configuration files. For more information, see "Using a configuration file to compile theme SWC files" on page 1568.

Using the include-file option to compile theme SWC files

The `include-file` option takes two arguments: a name and a path. The name is the name that you use to refer to that asset as in the CSS file. The path is the file system path to the asset. When using `include-file`, you are not required to add the resources to the source path. The following command line example includes the upIcon.jpg asset in the theme SWC file:

```
-include-file upIcon.jpg c:/myfiles/themes/assets/upIcon.jpg
```

You also specify the CSS file as a resource to include in the theme SWC file. You must include the .css extension when you provide the name of the CSS file; otherwise, Flex does not recognize it as a style sheet and does not apply it to your application.

You use the component compiler to compile theme assets, such as a style sheet and graphical skins, into a SWC file. The following example compiles a theme by using the compc command-line compiler:

```
compc -include-file mycss.css c:/myfiles/themes/mycss.css
    -include-file upIcon.jpg c:/myfiles/themes/assets/upIcon.jpg
    -include-file downIcon.jpg c:/myfiles/themes/assets/downIcon.jpg
    -include-file overIcon.jpg c:/myfiles/themes/assets/overIcon.jpg
    -o c:/myfiles/themes/MyTheme.swc
```

You cannot pass a list of assets to the `include-file` option. Each pair of arguments must be preceded by `-include-file`. Most themes use many skin files and style sheets, which can be burdensome to enter each time you compile. To simplify this, you can use the `load-config` option to specify a file that contains configuration options, such as multiple `include-file` options. For more information, see "Using a configuration file to compile theme SWC files" on page 1568.

Using the include-classes option to compile theme SWC files

The `include-classes` option takes a single argument: the name of the class to include in the SWC file. You pass the class name and not the class filename (for example, MyButtonSkin rather than MyButtonSkin.as). The class must be in your source path when you compile the SWC file.

The following command-line example compiles a theme SWC file that includes the CSS file and a single programmatic skin class, MyButtonSkin, which is in the themes directory:

```
compc -source-path c:/myfiles/flex/themes
    -include-file mycss.css c:/myfiles/flex/themes/mycss.css
    -include-classes MyButtonSkin -o c:/myfiles/flex/themes/MyTheme.swc
```

You can pass a list of classes to the `include-classes` option by space-delimiting each class, as the following example shows:

```
-include-classes MyButtonSkin MyControlBarSkin MyAccordionHeaderSkin
```

For more information about creating skin classes, see "Skinning MX components" on page 1655.

Using a configuration file to compile theme SWC files

Using a configuration file is generally more verbose than passing options on the command line, but it can make the component compiler options easier to read and maintain. For example, you could replace a command line with the following entries in a configuration file:

```
<?xml version="1.0"?>
<flex-config>
    <output>MyTheme.swc</output>
    <include-file>
        <name>mycss.css</name>
        <path>c:/myfiles/themes/mycss.css</path>
    </include-file>
    <include-file>
        <name>upIcon.jpg</name>
        <path>c:/myfiles/themes/assets/upIcon.jpg</path>
    </include-file>
    <include-file>
        <name>downIcon.jpg</name>
        <path>c:/myfiles/themes/assets/downIcon.jpg</path>
    </include-file>
    <include-file>
        <name>overIcon.jpg</name>
        <path>c:/myfiles/themes/assets/overIcon.jpg</path>
    </include-file>
    <include-classes>
        <class>MyButtonSkin</class>
        <class>MyAccordionHeaderSkin</class>
        <class>MyControlBarSkin</class>
    </include-classes>
</flex-config>
```

You can use the configuration file with compc by using the `load-config` option, as the following example shows:

```
compc -load-config myconfig.xml
```

You can also pass a configuration file to the Flash Builder component compiler. For more information on using the component compilers, see “Flex compilers” on page 2164.

Fonts

You can include fonts in your Adobe® Flex® applications. Although it is easier and more efficient to use the default device fonts, you can embed other fonts so that you can apply special effects to text-based controls, such as rotating and fading.

About fonts

When you compile an application in Flex, the application stores the names of the fonts that you used to create the text. Adobe® Flash® Player uses the font names to locate identical or similar fonts on the user’s system when the application runs. You can also embed fonts in the application so that the exact font is used, regardless of whether the client’s system has that font.

You define the font that appears in each of your components by using the `fontFamily` style property. You can set this property in an external style sheet, an `<fx:Style>` block, or inline. This property can take a list of fonts, as the following example shows:

```
.myClass {
    fontFamily: Arial, Helvetica;
    color: Red;
    fontSize: 22;
    fontWeight: bold;
}
```

If the client's system does not have the first font in the list, Flash Player attempts to find the second, and so on, until it finds a font that matches. If no fonts match, Flash Player makes a best guess to determine which font the client uses.

Fonts are inheritable style properties. So, if you set a font style on a container, all controls inside that container inherit that style, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/InheritableExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";

     s|VGroup {
        fontFamily: Helvetica;
        fontSize: 13pt;
     }

     s|HGroup {
        fontFamily: Times;
        fontSize: 13pt;
     }
  </fx:Style>
  <s:Panel title="Styles Inherited from VGroup Type Selector">
     <s:VGroup>
        <s:Button label="This Button uses Helvetica"/>
        <s:Label text="This Label control is in Helvetica."/>
        <s:RichText height="75" width="200">
```

```
            <s:text>
            The text in this RichText control uses the Helvetica font
            because it is inherited from the VGroup style.
            </s:text>
        </s:RichText>
    </s:VGroup>
  </s:Panel>
  <s:Panel title="Styles Inherited from HGroup Type Selector">
    <s:HGroup>
        <s:Button label="This Button uses Times"/>
        <s:Label text="This Label control is in Times."/>
        <s:RichText height="75" width="200">
            <s:text>
            The text in this RichText control uses the Times font
            because it is inherited from the HGroup style.
            </s:text>
        </s:RichText>
    </s:HGroup>
  </s:Panel>
</s:Application>
```

This example defines the HGroup and VGroup type selectors' `fontSize` and `fontFamily` properties. Flex applies these styles to all components in the container that support those properties; in these cases, the Button, Label, and RichText controls.

Controls that support FTE have limited font fallback. What this means is that when you try to use a special character that is not available in a font, Flash Player attempts to use a character from another font if possible.

Using device fonts

You can specify any font for the `fontFamily` property. However, not all systems have all font faces, which can result in an unexpected appearance of text in your application. The safest course when specifying font faces is to include a device font as a default at the end of the font list. *Device fonts* do not export font outline information and are not embedded in the SWF file. Instead, Flash Player uses whatever font on the client's local computer most closely resembles the device font.

Flash Player supports three device fonts. The following table describes these fonts:

Font name	Description
`_sans`	The `_sans` device font is a sans-serif typeface; for example, Helvetica or Arial.
`_serif`	The `_serif` device font is a serif typeface; for example, Times Roman.
`_typewriter`	The `_typewriter` device font is a monospace font; for example, Courier.

The following example specifies the device font `_sans` to use if Flash Player cannot find either of the other fonts on the client machine:

```
<?xml version="1.0"?>
<!-- fonts/DeviceFont.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!-- Use a vertical layout -->
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Style>
      @namespace s "library://ns.adobe.com/flex/spark";
     .myClass {
        fontFamily: Arial, Helvetica, "_sans";
        color: Red;
        fontSize: 12;
        fontWeight: bold;
     }
  </fx:Style>
  <s:Panel title="myClass Class Selector with Device Font">
     <s:VGroup styleName="myClass">
        <s:Button label="Click Me"/>
        <s:Label text="This is a Label control."/>
        <s:RichText width="200">
            <s:text>
            The text in the RichText control uses the myClass class selector.
            </s:text>
        </s:RichText>
     </s:VGroup>
  </s:Panel>
</s:Application>
```

***Note:** You must surround device font names with quotation marks when defining them with style declarations.*

Using device fonts does not affect the size of the SWF file because the fonts reside on the client. However, using device fonts can affect performance of the application because it requires that Flash Player interact with the local operating system. Also, if you use only device fonts, your selection is limited to three fonts.

Embed fonts

Rather than rely on a client machine to have the fonts you specify, you can embed fonts in your application. This means that the font is always available to Flash Player when the application is running, and you do not have to consider the implications of a missing font.

Spark components by default use the new Text Layout Framework (TLF) engine and require CFF embedded fonts. Renaun Erickson demonstrates (video) how to embed fonts for both TLF and non-TLF based components.

Peter DeHaan’s FlexExamples blog includes several examples of embedding fonts in Flex 4.

Cameron Yule explains how how embedding fonts in Flex 4 uses Text Layout Framework (TLF).

Supported file types include TrueType fonts (*.ttf), OpenType fonts (*.otf), as well as TrueType Collections (*.ttc), Mac Data Fork Fonts (*.dfont), and Mac Resource Fork TrueType Suitcases (which do not have a file extension).

***Note:** Check your font licenses before embedding any font files in your applications. Fonts might have licensing restrictions that preclude them from being stored as vector information.*

Benefits of embedded fonts

Embedded fonts have the following benefits:

- Client environment does not need the font to be installed.
- Embedded fonts can be rotated and faded.
- Embedded fonts are anti-aliased, which means that their edges are smoothed for easier readability. This is especially apparent when the text size is large.
- Embedded fonts provide smoother playback when zooming.
- Text appears exactly as you expect when you use embedded fonts.
- When you embed a font, you can use the advanced anti-aliasing information that provides clear, high-quality text rendering in SWF files. Using advanced anti-aliasing greatly improves the readability of text, particularly when it is rendered at smaller font sizes. For more information about advanced anti-aliasing, see "Using advanced anti-aliasing with non-CFF based fonts" on page 1579.

Drawbacks of using embedded fonts

Using embedded fonts is not always the best solution, however. Embedded fonts have the following limitations and drawbacks:

- Embed only TrueType or OpenType fonts and related "collection" formats. To embed other font types such as Type 1 PostScript fonts, embed that font in a SWF file that you create in Flash or with the fontswf utility, and then embed that SWF file in your application. For information about using the fontswf utility, see "Using the fontswf utility" on page 1584.
- Embedded fonts increase the file size of your application, because the document must contain font outlines for the text. This can result in longer download times for your users.
- In some cases, the text that is rendered by embedded fonts is truncated when they are used in visual components. This can happen, for example, when you explicitly set the width of a control. In these cases, you might be required to change the padding properties of the component by using style properties or subclassing it. This only occurs with some fonts.
- If you use MX controls in a Flex 4 application, you might have to add additional code to make the MX control use the embedded font. For more information, see "Embedding fonts with MX components" on page 1592.

Embed fonts with CSS

You typically use Cascading Style Sheets (CSS) syntax for embedding fonts in applications. You use the `@font-face` "at-rule" declaration to specify the source of the embedded font and then define the name of the font by using the `fontFamily` property. You typically specify the `@font-face` declaration for each face of the font for the same family that you use (for example, plain, bold, and italic).

Embed fonts with ActionScript

You can also embed fonts in ActionScript by using the `[Embed]` metadata tag. As with the `@font-face` declaration, you must specify a separate `[Embed]` tag for each font face.

Finding fonts

If you attempt to embed a font that the Flex compiler cannot find, Flex throws an error and your application does not compile. As a result, be sure to include directories that contain your fonts in the source path, or use absolute paths to the font file.

Embedded font syntax

To embed TrueType or OpenType fonts, you use the following syntax in your style sheet or `<fx:Style>` tag:

```
@font-face {
    src: url("location");
    fontFamily: alias;
    [fontStyle: normal | italic | oblique] ;
    [fontWeight: normal | bold | heavy] ;
    [embedAsCFF:true | false] ;
    [advancedAntiAliasing: true | false];
}
```

The following table describes the properties for the `@font-face` rule:

Property	Description
`src`	Specifies the file path location of a font. In Flex 4 and later, you cannot embed a font by its local name only. If you specify a relative location, the location is relative to the file in which the `@font-face` rule appears.
`fontFamily`	Sets the alias for the font that you use to apply the font in style sheets. This property is required. If you embed a font with a family name that matches the family name of a system font, the Flex compiler gives you a warning. You can disable this warning by setting the `show-shadows-system-font-warnings` compiler option to `false`.
`fontStyle`	Set the style type face value for the font. This property is optional, unless you are embedding a face that requires it. The default value is `normal`. Valid values are `normal`, `italic`, and `oblique`.

Property	Description
`fontWeight`	Set the weight type face value for the font. This property is optional, unless you are embedding a face that requires it. The default value is `normal`. Valid values are `normal`, `bold`, and `heavy`.
`embedAsCFF`	Indicates whether to embed an FTE-enabled font for components. Flash Text Engine (FTE) is a library that provides text controls with a rich set of formatting options. For Flex 4 and later, the default value is `true`. If you set the `compatibility-version` compiler option to 3.0.0, then the default value is `false`. If you set the `embedAsCFF` property to `true`, then you can use the advanced formatting features of FTE such as bidirectional text, kerning, and ligatures. If you set the value of `embedAsCFF` to `false`, then the embedded font does not support FTE, and works only with the MX text components. For information on using FTE-based classes for text rendering in your MX text controls, see "Embedding fonts with MX components" on page 1592.
`advancedAntiAliasing`	Determines whether to include the advanced anti-aliasing information when embedding the font. This property is optional and only used for legacy fonts. This property is ignored if you embed a font with the `embedAsCFF` property set to `true`. You cannot use this option when embedding fonts from a SWF file because this option requires access to the original, raw font file to pre-calculate anti-aliasing information at compile time. For more information on using advanced anti-aliasing, see "Using advanced anti-aliasing with non-CFF based fonts" on page 1579.

Example of @font-face rule

The following example embeds the MyriadWebPro.ttf font file:

```
@font-face {
    src: url("../assets/MyriadWebPro.ttf");
    fontFamily: myFontFamily;
    embedAsCFF: true;
}
```

After you embed a font with an `@font-face` declaration, you can use the value of the `fontFamily` property, or *alias*, in a selector's property declarations. The following example uses `myFontFamily`, the value of the `fontFamily` property, as the font in the VGroup type selector:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFace.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @namespace s "library://ns.adobe.com/flex/spark";
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontFamily;
        embedAsCFF: true;
     }

     s|VGroup {
        fontFamily: myFontFamily;
        fontSize: 15;
     }
  </fx:Style>
  <s:Panel title="Embedded Font Applied With Type Selector">
     <s:VGroup>
        <!-- This MX button tries to use a system font. -->
        <mx:Button label="Click Me"/>
        <!-- This Spark button uses the font of the VGroup container. -->
        <s:Button label="Click Me"/>
        <s:Label text="This is a Label control."/>
        <s:RichText width="250">
            <s:text>
                The text in this RichText control uses the
                font set on the VGroup.
            </s:text>
        </s:RichText>
     </s:VGroup>
  </s:Panel>
  <!-- This button uses the default font because it is not in the VGroup. -->
  <s:Button label="Click Me"/>

</s:Application>
```

Considerations when using the MX Button control

When you run this example, you might notice that the label on the MX Button (in the “mx” namespace) disappears. This is because the default style of an MX Button control’s label uses a bold typeface. However, the embedded font’s typeface (Myriad Web Pro) does not contain a definition for the bold typeface.

The Spark Button (in the “s” namespace) control’s label renders with the embedded font because it does not require a bold faced font.

To ensure that the MX Button control’s label uses the proper typeface, you can:

- Apply the MXFTEText.css theme if you mix MX and Spark components and use embedded fonts. In this case, you should embed both the plain and bold faces of the embedded font in separate `@font-face` rules. For more information, see “Using FTE in MX controls” on page 1594. This theme sets `embedAsCFF` to `false`.
- Add `fontWeight:bold` to the `@font-face` rule. This will render the MX Button label’s text, but with a device font.

- Embed a bold typeface so that the label of an MX Button control is rendered with the correct font. For information on embedding bold typefaces, see “Using multiple typefaces” on page 1585.
- Change the MX Button control’s label typeface to be non-bold. You can do this by creating a custom skin for the MX Button control.

Apply embedded font inline

You can also apply the embedded font inline by specifying the alias as the value of the control’s `fontFamily` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFaceInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontFamily;
        embedAsCFF: true;
     }
  </fx:Style>
  <s:Panel title="Embedded Font Applied Inline">
     <s:VGroup fontFamily="myFontFamily">
        <s:Button label="Click Me"/>
        <s:Label text="This is a Label."/>
        <s:RichText width="200" fontFamily="myFontFamily">
            <s:text>The text in the RichText control is Myriad Web Pro.</s:text>
        </s:RichText>
     </s:VGroup>
  </s:Panel>
</s:Application>
```

Locating embedded fonts

The `src` property in the `@font-face` declaration specifies the location of the font family. You use the `src` property to embed a TrueType or OpenType font by location by specifying a valid URI to the font. The URI can be relative (for example, ../fontfolder/akbar.ttf) or absolute (for example, c:/myfonts/akbar.ttf). The URI can also point to a SWF file that has embedded fonts within it, such as a SWF file created with the fontswf utility. For information about using the fontswf utility, see “Using the fontswf utility” on page 1584.

You must specify the `url` of the `src` property in the `@font-face` declaration. All other properties are optional.

Embedding fonts in ActionScript

You can embed TrueType or OTF font files by location by using the `[Embed]` metadata tag in ActionScript. To embed a font by location, you use the `source` property in the `[Embed]` metadata tag.

The `[Embed]` metadata tag takes the same properties that you set as the @font-face rule. You separate them with commas. For the list of properties, see “Embedded font syntax” on page 1573

The following examples embed fonts by location by using the `[Embed]` tag syntax:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFaceActionScriptByLocation.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="700">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     .mystyle1 {
        fontFamily:myMyriadFont;
        fontSize: 32pt;
     }
     .mystyle2 {
        fontFamily:myBoldMyriadFont;
        fontSize: 32pt;
        fontWeight: bold;
     }
  </fx:Style>

  <fx:Script>
     /*
      * Embed a font by location.
      */
     [Embed(source='../assets/MyriadWebPro.ttf',
        fontName='myMyriadFont',
        mimeType='application/x-font',
        embedAsCFF='true'
     )]
     // You do not use this variable directly. It exists so that
     // the compiler will link in the font.
     private var font1:Class;

     /*
      * Embed a font with bold typeface by location.
      */
     [Embed(source='../assets/MyriadWebPro-Bold.ttf',
        fontWeight='bold',
        fontName='myBoldMyriadFont',
        mimeType='application/x-font',
        embedAsCFF='true'
     )]
     private var font2:Class;

  </fx:Script>
  <s:Panel title="Embedded Fonts Using ActionScript">
     <s:VGroup>
        <s:RichText
            width="100%"
            height="75"
            styleName="mystyle1"
            text="This text uses the MyriadWebPro font."
        />
        <s:RichText
            width="100%"
            height="75"
            styleName="mystyle2"
            text="This text uses the MyriadWebPro-Bold font."
        />
     </s:VGroup>
  </s:Panel>
</s:Application>
```

You use the value of the `fontName` property that you set in the `[Embed]` tag as the alias (`fontFamily`) in your style definition.

Note that specifying the `mimeType` is not necessary if your font uses a known file extension such as *.ttf.

To embed a font with a different typeface (such as bold or italic), you specify the `fontWeight` or `fontStyle` properties in the `[Embed]` statement and in the style definition. For more information on embedding different typefaces, see "Using multiple typefaces" on page 1585.

You can specify a subset of the font's character range by specifying the `unicodeRange` parameter in the `[Embed]` metadata tag or the `@font-face` declaration. Embedding a range of characters rather than using the default of all characters can reduce the size of the embedded font and, therefore, reduce the final output size of your SWF file. For more information, see "Setting character ranges" on page 1588.

Embedding fonts in container formats

When the `@font-face src` property points to a file that is a "container" of several fonts (such as a *.ttc or *.dfont file), you use the `fontFamily` property to select the precise font face you want out of the collection. The value of the `fontFamily` property should be the full font name of the exact font face that you want to target.

The following example embeds and uses two different fonts from the same TTC file:

```
<?xml version="1.0"?>
<!-- fonts/TTCTest.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @font-face {
            src:url("../assets/CAMBRIA.TTC");
            fontFamily: "Cambria Math";
            embedAsCFF: true;
        }
        @font-face {
            src:url("../assets/CAMBRIA.TTC");
            fontFamily: "Cambria";
            embedAsCFF: true;
        }
        .useCambriaMath {
            fontFamily: "Cambria Math";
            fontSize:24;
        }
        .useCambria {
            fontFamily: "Cambria";
            fontSize:24;
        }
    </fx:Style>
    <s:HGroup>
        <s:Button label="Default Font" fontSize="24"/>
        <s:Button label="Cambria Math" styleName="useCambriaMath"/>
        <s:Button label="Cambria" styleName="useCambria"/>
    </s:HGroup>
</s:Application>
```

You can also use this technique when using the `[Embed]` syntax.

Using advanced anti-aliasing with non-CFF based fonts

Flex 3 components in the Halo theme use non-CFF fonts. The text in these components is rendered with the flash.text.TextField API (instead of the new Flash Text Engine (FTE).

When you embed non-CFF fonts (with the `embedAsCFF` property set to `false`), you can use advanced anti-aliasing to provide those fonts with additional information about the font. Non-CFF embedded fonts that use the advanced anti-aliasing information are typically clearer and appear sharper at smaller font sizes. CFF fonts have this information by default.

By default, non-CFF fonts that you embed in applications use the advanced anti-aliasing information. This default is set by the `fonts.advanced-anti-aliasing` compiler option in the flex-config.xml file (the default value is `true`). You can override this default value by setting the value in your style sheets or changing it in the configuration file. To disable advanced anti-aliasing in style sheets, you set the `advancedAntiAliasing` style property to `false` in your `@font-face` rule, as the following example shows:

```
@font-face {
    src:url("../assets/MyriadWebPro.ttf");
    fontFamily: myFontFamily;
    advancedAntiAliasing: false;
    embedAsCFF: false;
}
```

Using advanced anti-aliasing can degrade the performance of your compiler. This is not a run-time concern, but can be noticeable if you compile your applications frequently or use the web-tier compiler. Using advanced anti-aliasing can also cause a slight delay when you load SWF files. You notice this delay especially if you are using several different character sets, so be aware of the number of fonts that you use. The presence of advanced anti-aliasing information may also cause an increase in the memory usage in Flash Player and Adobe® AIR™. Using four or five fonts, for example, can increase memory usage by approximately 4 MB.

When you embed non-CFF fonts that use advanced anti-aliasing in your applications, the fonts function exactly as other embedded fonts. They are anti-aliased, you can rotate them, and you can make them partially or wholly transparent.

Font definitions that use advanced anti-aliasing support several additional styles properties: `fontAntiAliasType`, `fontGridFitType`, `fontSharpness`, and `fontThickness`. These properties are all inheriting styles, but they are applied on the component being styled. They are not relevant to the actual font embedding process (and thus must not be specified in the `@font-face` rule).

Because the advanced anti-aliasing-related style properties are CSS styles, you can use them in the same way that you use standard style properties, such as `fontFamily` and `fontSize`. For example, a text-based component could use subpixel-fitted advanced anti-aliasing of New Century 14 at sharpness 50 and thickness -35, while all Button controls could use pixel-fitted advanced anti-aliasing of Tahoma 10 at sharpness 0 and thickness 0. These styles apply to all the text in a TextField control; you cannot apply them to some characters and not others.

The default values for the advanced anti-aliasing styles properties are defined in the defaults.css file. If you replace this file or use another style sheet that overrides these properties, Flash Player and AIR use the standard font renderer to render the fonts that use advanced anti-aliasing. If you embed fonts that use advanced anti-aliasing, you must set the `fontAntiAliasType` property to `advanced`, or you lose the benefits of the advanced anti-aliasing information.

The following table describes these properties:

Style property	Description
fontAntiAliasType	Sets the antiAliasType property of internal TextField controls. The valid values are normal and advanced. The default value is advanced, which enables advanced anti-aliasing for the font. Set this property to normal to prevent the compiler from using advanced anti-aliasing. This style has no effect for system fonts or fonts embedded without the advanced anti-aliasing information.
fontGridFitType	Sets the gridFitType property of internal TextField controls. The valid values are none, pixel, and subpixel. The default value is pixel. For more information, see the TextField and GridFitType classes in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. This property has the same effect as the gridFitType style property of the TextField control for system fonts, only it applies when you embed fonts with advanced anti-aliasing. Changing the value of this property has no effect unless the fontAntiAliasType property is set to advanced.
fontSharpness	Sets the sharpness property of internal TextField controls. The valid values are numbers from -400 to 400. The default value is 0. This property has the same effect as the fontSharpness style property on the TextField control for system fonts, only it applies when you embed fonts with advanced anti-aliasing. Changing the value of this property has no effect unless the fontAntiAliasType property is set to advanced.
fontThickness	Sets the thickness property of internal TextField controls. The valid values are numbers from -200 to 200. The default value is 0. This property has the same effect as the fontThickness style property on the TextField control for system fonts, only it applies when you embed fonts with advanced anti-aliasing. Changing the value of this property has no effect unless the fontAntiAliasType property is set to advanced.

To use functionality similar to advanced anti-aliasing with CFF based fonts, you use the functionality of FTE that is built into Spark's text-based controls. For more information, see "Formatting Data" on page 2004.

Detecting embedded fonts

You can use the SystemManager class's isFontFaceEmbedded() method to determine whether the font is embedded or whether it has been registered globally with the register() method of the Font class. The isFontFaceEmbedded() method takes a single argument—the object that describes the font's TextFormat—and returns a Boolean value that indicates whether the font family you specify is embedded, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/DetectingEmbeddedFonts.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
     creationComplete="determineIfFontFaceIsEmbedded()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Style>
     @font-face {
        src: url(../assets/MyriadWebPro.ttf);
        fontFamily: myPlainFont;
        embedAsCFF: true;
     }

     .myStyle1 {
        fontFamily: myPlainFont;
        fontSize:12pt
     }
  </fx:Style>

  <fx:Script><![CDATA[
     import mx.managers.SystemManager;
     import mx.core.FlexGlobals;
     import flash.text.TextFormat;

    [Bindable]
     private var b1:Boolean;
    [Bindable]
     private var b2:Boolean;

     public function determineIfFontFaceIsEmbedded():void {
        var tf1:TextFormat = new TextFormat();
        tf1.font = "myPlainFont";

        var tf2:TextFormat = new TextFormat();
        tf2.font = "Arial";

        b1 = FlexGlobals.topLevelApplication.systemManager.
           isFontFaceEmbedded(tf1);
        b2 = FlexGlobals.topLevelApplication.systemManager.
           isFontFaceEmbedded(tf2);
     }
  ]]></fx:Script>
  <s:Form>
     <s:FormItem label="isFontFaceEmbedded (myPlainFont):">
        <s:Label id="l1" text=" {b1}"/>
     </s:FormItem>
     <s:FormItem label="isFontFaceEmbedded (Arial):">
        <s:Label id="l2" text="{b2}"/>
     </s:FormItem>
  </s:Form>
</s:Application>
```

In this example, the font identified by the myPlainFont family name is embedded, but the Arial font is not.

You can use the Font class's `enumerateFonts()` method to output information about device or embedded fonts. The following example lists embedded fonts:

```
<?xml version="1.0"?>
<!-- fonts/EnumerateFonts.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="listFonts()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFont;
        embedAsCFF: true;
     }
     @font-face {
        src:url("../assets/MyriadWebPro-Bold.ttf");
        fontFamily: myFont;
        fontWeight: bold;
        embedAsCFF: true;
     }
     @font-face {
        src:url("../assets/MyriadWebPro-Italic.ttf");
        fontFamily: myFont;
        fontStyle: italic;
        embedAsCFF: true;
     }

     .myPlainStyle {
        fontSize: 20;
        fontFamily: myFont;
     }

     .myBoldStyle {
        fontSize: 20;
        fontFamily: myFont;
        fontWeight: bold;
     }
     .myItalicStyle {
        fontSize: 20;
        fontFamily: myFont;
        fontStyle: italic;
     }
  </fx:Style>
  <fx:Script><![CDATA[
     private function listFonts():void {
        var fontArray:Array = Font.enumerateFonts(true);
        ta1.text += "Fonts: \n";
        for(var i:int = 0; i < fontArray.length; i++) {
            var thisFont:Font = fontArray[i];
            ta1.text += "FONT " + i + ":: name: " + thisFont.fontName + "; typeface: " +
                thisFont.fontStyle + "; type: " + thisFont.fontType;
            if (thisFont.fontType == "embeddedCFF"||thisFont.fontType == "embedded") {
                ta1.text += "*";
            }
            ta1.text +=  "\n";

        }

    }
  ]]></fx:Script>
  <s:VGroup>
     <s:RichText text="Plain Label" styleName="myPlainStyle"/>
     <s:RichText text="Bold Label" styleName="myBoldStyle"/>
     <s:RichText text="Italic Label" styleName="myItalicStyle"/>
     <s:TextArea id="ta1" height="200" width="400"/>
     <s:RichText text="* Embedded" styleName="myItalicStyle"/>
  </s:VGroup>
</s:Application>
```

The following list shows the first few lines of sample output. This list will vary depending on the client's system.

```
FONT 0:: name: myFont; typeface: regular; type: embeddedCFF*
FONT 1:: name: myFont; typeface: bold; type: embeddedCFF*
FONT 2:: name: myFont; typeface: italic; type: embeddedCFF*
FONT 3:: name: Marlett; typeface: regular; type: device
FONT 4:: name: Arial; typeface: regular; type: device
FONT 5:: name: Arial CE; typeface: regular; type: device
```

The `enumerateFonts()` method takes a single Boolean argument: `enumerateDeviceFonts`. The default value of the `enumerateDeviceFonts` property is `false`, which means it returns an Array of embedded fonts by default.

If you set the `enumerateDeviceFonts` argument to `true`, the `enumerateFonts()` method returns an array of available device fonts on the client system, but only if the client's mms.cfg file sets the `DisableDeviceFontEnumeration` property to 0, the default value. If you set the `DisableDeviceFontEnumeration` property to 1, Flash Player cannot list device fonts on a client computer unless you explicitly configure the client to allow it. For more information about configuring the client with the mms.cfg file, see the Flash Player documentation.

Using the fontswf utility

The fontswf utility is a simple command line tool that converts a single font face from a font file into a SWF file. This SWF file can be used as the source of an embedded font in your applications. Supported font file types are *.ttf, *.otf, *.ttc, and *.dfont.

The fontswf utility is for users of the Mozilla Public License version of the Flex SDK. This version includes only open-source technology. Because the font managers are not open-source, this utility can be used in their place so that you can embed fonts in your applications.

The fontswf utility is in the *sdk_root*/bin directory of the Flex SDK. For Flash Builder, the fontswf utility is located in the sdks/4.6.0/bin directory. If you do not have the fontswf utility in your bin directory, you must get a more recent version of the SDK.

You use fontswf by invoking it from the command line, as the following example shows:

```
fontswf [options] font_input_file
```

For example:

```
c:\flex\bin> fontswf -4 -u U+0020-007F -bold -o c:/temp/myboldfont.swf
c:/assets/fonts/myboldfont.ttf
```

The following table describes the options for the fontswf utility:

Option	Description
`-a, -alias` ***`name`***	Sets the font’s alias. The default is the font’s family name.
`-b, -bold`	Embeds the font’s bold face.
`-i, -italic`	Embeds the font’s italic face.
`-o, -output` ***`file_path`***	Sets the output file path for the SWF file.
`-u, -unicode-range` ***`range`***	Sets the included character range. The default value is “*”, which includes all characters. For information on using character ranges, see “Setting character ranges” on page 1588.
`-3`	Generates a font SWF file for applications that use TextField-based text rendering. Use this option if you are creating a font SWF file for a Flex 3 application.
`-4`	Generates a font SWF file for applications that support CFF (Flex 4) with Flash Player 10. This is the default option.

Using multiple typefaces

Most fonts have four typeface styles: plain, bold, italic, and bold-italic. You can embed any number of these typeface styles for each font in your applications. If you embed only the bold typeface in your application, you cannot use the normal (or plain) typeface unless you also embed that typeface. For each typeface that you use, you must add a new `@font-face` declaration to your style sheet.

***Note:** Some Flex controls, such as the MX Button control, use the bold typeface style by default, rather than the plain style. If you use an embedded font for an MX Button’s label, you must either embed the bold font style for that font, or set the default typeface for the MX Button label to match a typeface that you embed.*

The following example embeds the bold, italic, and plain typefaces of the Myriad Web Pro font. After you define the font face, you define selectors for the font by using the same alias as the `fontFamily`. You define one for the bold, one for the italic, and one for the plain face. To apply the font styles, this example applies the class selectors to the Label controls inline:

```
<?xml version="1.0"?>
<!-- fonts/MultipleFaces.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!-- Use a vertical layout -->
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFont;
        embedAsCFF: true;
     }
     @font-face {
        /* Note the different filename for boldface. */
        src:url("../assets/MyriadWebPro-Bold.ttf");
        fontFamily: myFont; /* Notice that this is the same alias. */
        fontWeight: bold;
        embedAsCFF: true;
     }
     @font-face {
        /* Note the different filename for italic face. */
        src:url("../assets/MyriadWebPro-Italic.ttf");
        fontFamily: myFont; /* Notice that this is the same alias. */
        fontStyle: italic;
        embedAsCFF: true;
     }

     .myPlainStyle {
        fontSize: 32;
        fontFamily: myFont;
     }

     .myBoldStyle {
        fontSize: 32;
        fontFamily: myFont;
        fontWeight: bold;
     }
     .myItalicStyle {
        fontSize: 32;
        fontFamily: myFont;
        fontStyle: italic;
     }
  </fx:Style>
  <s:VGroup>
     <s:Label text="Plain Text" styleName="myPlainStyle"/>
     <s:Label text="Italic Text" styleName="myItalicStyle"/>
     <s:Label text="Bold Text" styleName="myBoldStyle"/>
  </s:VGroup>
</s:Application>
```

Optionally, you can apply the bold or italic type to controls inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/MultipleFacesAppliedInline.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFont;
        embedAsCFF: true;
     }
     @font-face {
        src:url("../assets/MyriadWebPro-Bold.ttf");
        fontFamily: myFont;
        fontWeight: bold;
        embedAsCFF: true;
     }
     @font-face {
        src:url("../assets/MyriadWebPro-Italic.ttf");
        fontFamily: myFont;
        fontStyle: italic;
        embedAsCFF: true;
     }

     .myStyle1 {
        fontSize: 32;
        fontFamily: myFont;
     }
  </fx:Style>
  <s:VGroup styleName="myStyle1">
     <s:Label text="Plain Text"/>
     <s:Label text="Italic Text" fontStyle="italic"/>
     <s:Label text="Bold Text" fontWeight="bold"/>
  </s:VGroup>
</s:Application>
```

If you use a bold-italic font, the font must have a separate typeface for that font. You specify both properties (`fontWeight` and `fontStyle`) in the `@font-face` and selector blocks, as the following example shows:

```
@font-face {
    src:url("../assets/KNIZIA-BI.TTF");
    fontStyle: italic;
    fontWeight: bold;
    fontFamily: myFont;
    embedAsCFF: true;
}
.myBoldItalicStyle {
    fontFamily:myFont;
    fontWeight:bold;
    fontStyle:italic;
    fontSize: 32;
}
```

In the `@font-face` definition, you can specify whether the font is bold or italic by using the `fontWeight` and `fontStyle` properties. For a bold font, you can set `fontWeight` to `bold` or an integer greater than or equal to 700. You can specify the `fontWeight` as `plain` or `normal` for a nonboldface font. For an italic font, you can set `fontStyle` to `italic` or `oblique`. You can specify the `fontStyle` as `plain` or `normal` for a nonitalic face. If you do not specify a `fontWeight` or `fontStyle`, Flex assumes you embedded the plain or regular font face.

You can also add any other properties for the embedded font, such as `fontSize`, to the selector, as you would with any class or type selector.

Some fonts with multiple faces are packaged as TTC (TrueType Collection) files. In these cases, you can use the `fontFamily` alias to embed the specific face in the TTC file that you want. For more information, see "Embedding fonts in container formats" on page 1578.

By default, Flex includes the entire font definition for each embedded font in the application, so you should limit the number of fonts that you use to reduce the size of the application. You can limit the size of the font definition by defining the character range of the font. For more information, see "Setting character ranges" on page 1588.

Setting character ranges

By specifying a range (or subset) of characters that compose the face of an embedded font, you reduce the size of an embedded font. Each character in a font that you use must be described; removing some of these characters reduces the overall size of the description information that Flex must include for each embedded font.

You can set the range of glyphs in the flex-config.xml file or in the `@font-face` declaration. You specify individual characters or ranges of characters using the Unicode values for the characters, and you can set multiple ranges for each font declaration.

The syntax for setting a character range is as follows:

```
U+[beginning of range]-[end of range];
```

For example:

```
U+0041-005A
```

You can also use wild cards Unicode range syntax; for example:

```
U+00??
```

This is the same as `U+0000-00FF`.

You can also refer to a single character without a range; for example:

```
U+0041
```

Finally, you can specify a list of ranges by using a comma separator; for example:

```
U+0041,U+0043-00FF,U+0045
```

If you use a character that is outside of the declared range, Flex displays a device font for that character (for FTE-based controls) or a blank character (for non-FTE-based controls). For more information on setting character ranges in applications built with Flex, see the CSS-2 Fonts specification at www.w3.org/TR/1998/REC-CSS2-19980512/fonts.html#descdef-unicode-range.

If you embed a font from a SWF file that you create with Flash or the fontswf utility, you should set the character range to include only those characters that you need to keep the size of the SWF file as small as possible. For information about using the fontswf utility, see "Using the fontswf utility" on page 1584.

Setting ranges in font-face declarations

You can set the range of embedded characters in an application by using the `unicodeRange` property of the `@font-face` declaration. The following example embeds the Myriad Web Pro font and defines the range of characters for the font in the `<fx:Style>` tag:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontCharacterRange.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
      @namespace s "library://ns.adobe.com/flex/spark";
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontFamily;
        embedAsCFF: true;
        unicodeRange:
           U+0041-005A, /* Upper-Case [A..Z] */
           U+0061-007A, /* Lower-Case a-z */
           U+0030-0039, /* Numbers [0..9] */
           U+002E-002E; /* Period [.] */
     }
     s|RichText {
        fontFamily: myFontFamily;
        fontSize: 32;
     }
  </fx:Style>
  <s:Panel title="Embedded Font Character Range">
        <s:RichText width="400" height="150">
          <s:text>The Text Uses Only Some of Available Characters 0 1 2 3 4 5 6 7 8 9.</s:text>
        </s:RichText>
  </s:Panel>
</s:Application>
```

Setting language ranges in flex-config.xml

You can specify the language and character range for embedded fonts in the flex-config.xml file by using the `<language-range>` child tag. This lets you define the range once and use it across multiple `@font-face` blocks.

The following example creates named ranges in the flex-config.xml file:

```
<fonts>
    <languages>
        <language-range>
            <lang>englishRange</lang>
            <range>U+0020-007E</range>
        </language-range>
        <language-range>
            <lang>lowerCaseEnglish</lang>
            <range>U+0061-007A</range>
        </language-range>
    <languages>
</fonts>
```

In your style sheet, you point to the defined ranges by using the `unicodeRange` property of the `@font-face` declaration, as the following example shows:

```
@font-face {
    fontFamily: myPlainFont;
    src: url("../assets/MyriadWebPro.ttf");
    unicodeRange: "englishRange";
    embedAsCFF: true;
}
```

Flex includes a file that lists convenient mappings of the Flash UnicodeTable.xml character ranges for use in the Flex configuration file. For the Flex SDK, the file is located at *flex_install_dir*/frameworks/flash-unicode-table.xml. For Adobe LiveCycle Data Services ES, the file is located at *flex_app_root*/WEB-INF/flex/flash-unicode-table.xml.

The following example shows the predefined range Latin 1:

```
<language-range>
    <lang>Latin I</lang>
    <range>U+0020,U+00A1-00FF,U+2000-206F,U+20A0-20CF,U+2100-2183</range>
</language-range>
```

To make ranges listed in the flash-unicode-table.xml file available in your applications, copy the ranges from this file and add them to the flex-config.xml files.

Detecting available ranges

You can use the Font class to detect the available characters in an embedded font. You do this with the `hasGlyphs()` method.

The following example embeds the same font twice, each time restricting the font to different character ranges. The first font includes support only for the letters A and B. The second font family includes all 128 glyphs in the Basic Latin block.

```
<?xml version="1.0"?>
<!-- charts/CharacterRangeDetection.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="checkCharacterSupport();"
>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @font-face {
            font-family: myABFont;
            src:url("../assets/MyriadWebPro.ttf");
            /*
            * Limit range to the characters A and B.
            */
            unicodeRange: U+0041-0042;
            embedAsCFF: true;
        }
        @font-face {
            font-family: myWideRangeFont;
            src:url("../assets/MyriadWebPro.ttf");
            /*
            * Set range to the 128 characters in
            * the Basic Latin block.
            */
            unicodeRange: U+0041-007F;
            embedAsCFF: true;
        }
    </fx:Style>
    <fx:Script><![CDATA[
        public function checkCharacterSupport():void {
            var fontArray:Array = Font.enumerateFonts(false);
            for(var i:int = 0; i < fontArray.length; i++) {
                var thisFont:Font = fontArray[i];
                if (thisFont.hasGlyphs("DHARMA")) {
                    ta1.text += "The font '" + thisFont.fontName +
                        "' supports these glyphs.\n";
                } else {
                    ta1.text += "The font '" + thisFont.fontName +
                        "' does not support these glyphs.\n";
                }
            }
        }
    ]]></fx:Script>
    <s:VGroup>
        <s:RichText>
            <s:text>myABFont unicodeRange: U+0041-0042 (characters A and B)</s:text>
        </s:RichText>
        <s:RichText>
            <s:text>myWideRangeFont unicodeRange: U+0041-007F (Basic Latin chars)</s:text>
        </s:RichText>
        <s:Label text="Glyphs: DHARMA"/>
        <s:RichText id="ta1" height="150" width="300"/>
    </s:VGroup>
</s:Application>
```

Embedding multi-byte fonts

When using multi-byte fonts that have a large number of characters, such as those used in Asian languages, you should embed the smallest possible set of characters. If you embed a font's entire character set, the size of your application's SWF file can be very large. You can define sets of Unicode character ranges in the flex-config.xml file and then reference the name of that range in your style's `@font-face` declaration.

The Flex CookBook shows you how to limit character sets when embedding fonts.

Flex provides predefined character ranges for common multi-byte scripts such as Thai, Kanji, Hangul, and Hebrew in the flash-unicode-table.xml file. This file is not processed by Flex, but is included to provide you with ready definitions for various character ranges. For example, the following character range for Thai is listed in the flash-unicode-table.xml file:

```
<language-range>
    <lang>Thai</lang>
    <range>U+0E01-0E5B</range>
</language-range>
```

To use this language in your application, copy the character range to the flex-config.xml file or pass it on the command line by using the `fonts.languages.language-range` option. Add the full definition as a child tag to the `<languages>` tag, as the following example shows:

```
<flex-config>
    <compiler>
        <fonts>
            <languages>
                <language-range>
                    <lang>thai</lang>
                    <range>U+0E01-0E5B</range>
                </language-range>
            </languages>
        </fonts>
    </compiler>
    ...
</flex-config>
```

You can change the value of the `<lang>` element to anything you want. When you embed the font by using CSS, you refer to the language by using this value in the `unicodeRange` property of the `@font-face` declaration, as the following example shows:

```
@font-face {
    fontFamily:"Thai_font";
    src: url("../assets/THAN.TTF"); /* Embed from file */
    unicodeRange:"thai";
    embedAsCFF: true;
}
```

Embedding fonts with MX components

The `embedAsCFF` (Compact Font Format) property indicates whether to embed a font that supports the advanced text layout features used by the Flash Text Engine (FTE). This is sometimes referred to as DefineFont4. If you set the value of the `embedAsCFF` property to `true`, then you can only use that font with controls that support FTE. If you set the value of the `embedAsCFF` property to `false`, then the embedded font does not support FTE and you can only use that font with controls that do not have FTE support.

The implications of this appear when you try to use MX controls with a font that was embedded with FTE support. In those cases, the text does not appear, as the following example shows.

In the first panel, the embedded CFF font is applied to the Spark control and the embedded non-CFF font is applied to the MX control. The result is that the Button labels show correctly. In the second Panel, both Button labels use the CFF font. The result is that no text appears for the MX control in the second Panel because the MX control does not support CFF.

```
<?xml version="1.0"?>
<!-- fonts/CFFTest.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myCFFFont;
        embedAsCFF: true;
     }
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontNoCFF;
        embedAsCFF: false;
     }

     .myCFFStyle {
        fontSize: 32;
        fontFamily: myCFFFont;
     }
     .myStyleNoCFF {
        fontSize: 32;
        fontFamily: myFontNoCFF;
     }
  </fx:Style>
  <s:Panel title="Using Correct Fonts">
    <s:VGroup>
        <s:Button label="Spark Button" styleName="myCFFStyle"/>
        <mx:Button label="MX Button" styleName="myStyleNoCFF"/>
    </s:VGroup>
  </s:Panel>
  <s:Panel title="Using Incorrect Fonts">
    <s:VGroup>
        <s:Button label="Spark Button" styleName="myCFFStyle"/>
        <!-- The Button label will not show up because it's a MX control that
        is attempting to use a Spark-compatible embedded font. -->
        <mx:Button label="MX Label" styleName="myCFFStyle"/>
    </s:VGroup>
  </s:Panel>
</s:Application>
```

As this example illustrates, if you mix MX and Spark controls inside the same application, you might not be able to use the same embedded font.

There are two possible remedies to this situation:

- Specify that the MX controls use FTE classes to render text, rather than their default text renderers.
- Embed both the non-CFF version of the font in addition to the CFF version of the font. This is the less-desireable approach because it increases the size of your SWF file.

The following sections describe each of these solutions.

More Help topics

"MX text controls" on page 805

Using FTE in MX controls

The controls that support FTE include all Spark components in the spark.components.* package. This includes the Spark text controls such as Label, RichText, and RichEditableText. This also includes Spark versions of the TextInput and TextArea controls. This does not include MX controls in the mx.controls.* package.

The reason that MX controls do not support FTE is that in previous versions of Flex, they used the UITextField subcomponent to render text. This subcomponent does not support FTE. Spark controls, on the other hand, use FTE-compatible classes to render text.

The Flex SDK provides the mx.core.UIFTETextField and mx.controls.MXFTETextInput classes that support FTE for MX text controls. You can use these classes in some MX controls so that those controls can use CFF versions of embedded fonts. As a result, those controls can use the same embedded fonts that you also use with the Spark controls. You do this by setting the `textFieldClass` and `textInputClass` styles to use these classes.

The easiest way to use the MXFTETextInput and UIFTETextField classes with your MX text controls is to apply the MXFTEText.css theme file to your application. This theme causes your MX controls to use the MXFTETextInput and UIFTETextField classes for text rendering. The MXFTEText.css theme file is a convenience theme that is set up to apply only FTE-supporting classes to MX controls.

The following excerpt from the MXFTEText.css theme file shows that the `textInputClass` and `textFieldClass` style properties are set to classes that support FTE:

```
DateField {
    textInputClass: ClassReference("mx.controls.MXFTETextInput");
}
Label {
    textFieldClass: ClassReference("mx.core.UIFTETextField");
}
```

On the command line, you specify the MXFTEText.css theme file by using the `theme` compiler option, as the following example shows:

```
mxmlc -theme+=frameworks/projects/spark/MXFTEText.css MyApp.mxml
```

Note that instead of setting `theme=`*`filename`*, this example uses `theme+=`*`filename`*. This is so that the MXFTEText.css file is used as a theme, *in addition* to the default themes, not as a replacement of the default themes.

In Flash Builder, you can use the MXFTEText.css theme file in your application by selecting Project > Properties > Flex Compiler. Then select the Use Flash Text Engine in MX Components option.

The following example is compiled with the the MXFTEText.css theme. Because this theme is used, the MX text controls can use the same embedded font as the Spark embedded font. If you compile this example without the `theme` option, the label for the MX Button does not render.

```
<?xml version="1.0"?>
<!-- fonts/UseTLFTextTheme.mxml -->
<!-- Compile this example by setting theme=MXFTEText.css for a compiler argument. -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myCFFFont;
        embedAsCFF: true;
     }
     .myCFFStyle {
        fontSize: 32;
        fontFamily: myCFFFont;
     }
  </fx:Style>
  <s:Panel title="Spark and MX Buttons">
    <s:VGroup>
        <s:Button label="Spark Button" styleName="myCFFStyle"/>
        <mx:Button label="MX Button" styleName="myCFFStyle"/>
    </s:VGroup>
  </s:Panel>
</s:Application>
```

The MX DataGrid control has a special class, FTEDataGridItemRenderer, that you can use for custom item renderers. The MXFTEText.css theme file specifies it as follows:

```
defaultDataGridItemRenderer:
ClassReference("mx.controls.dataGridClasses.FTEDataGridItemRenderer");
```

Some controls are not affected by the MXFTEText.css theme. This is because some MX controls have Spark equivalents, such as Button or ComboBox. As a result, you should use the Spark version of the control instead of the MX version where possible.

Rather than use the MXFTEText.css theme file to add FTE support to your MX controls, you can manually replace the non-FTE classes with the FTE classes for text rendering on a MX control. You do this by setting the value of the control's `textFieldClass` or `textInputClass` style properties to the UIFTETextField or MXFTETextInput classes, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fonts/TextFieldClassExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout
            paddingLeft="20" paddingRight="20"
            paddingTop="20" paddingBottom="20" />
    </s:layout>
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        @namespace s "library://ns.adobe.com/flex/spark";
        @font-face {
            src:url("../assets/MyriadWebPro.ttf");
            fontFamily: cffFont;
            embedAsCFF: true;
        }
        mx|Label{
            fontFamily: cffFont;
            fontSize: 22;
            textFieldClass:ClassReference("mx.core.UIFTETextField");
        }
        s|Label{
            fontFamily: cffFont;
            fontSize: 22;
        }
    </fx:Style>
    <mx:Label text="Hello World 1234567890 [MX Label]" width="100%"/>
    <s:Label text="Hello World 1234567890 [Spark Label]" width="100%"/>
</s:Application>
```

You can also specify the value of the `textFieldClass` or `textInputClass` style properties inline, as the following example shows:

```
<mx:Label text="Hello World 1234567890 [Label]" width="100%"
textFieldClass="mx.core.UIFTETextField"/>
```

Do not use the `setStyle()` method to specify the value of these style properties in ActionScript.

The UIFTETextField class does not support editing, scrolling, selection, linking, or rich text. As a result, you can only use this class on MX controls that do not use these features. The MXFTETextInput class can be edited and is selectable.

The following MX controls support the UIFTETextField and MXFTETextInput classes, which means that you can use CFF fonts with them:

- Accordion
- Alert (the text is not selectable)
- Button
- ButtonBar
- CheckBox
- DateChooser
- FileSystemComboBox

- FileSystemHistoryButton
- FormHeading
- FormItem
- FormHeading
- FormItem
- HSlider (the labels follow the same rules as the MX Label control)
- LinkBar
- LinkButton
- Menu
- MenuBar
- Panel
- PopUpButton
- PopUpMenuButton
- PrintDataGrid
- ProgressBar
- RadioButton
- TabBar
- TabNavigator
- TitleWindow
- ToggleButtonBar
- ToolTip
- VSlider (the labels follow the same rules as the MX Label control)

Other MX controls have some limitations when it comes to using the UIFTETextField and MXFTETextInput classes. In some cases, you should use the Spark equivalents. In other cases, you can use the UIFTETextField and MXFTETextInput subcomponents with the MX control, as long as you avoid using advanced text features, such as editability or the `htmlText` property. The following table describes these limitations:

MX control	Description
List-based components (such as List, FileSystemList, HorizontalList, TileList, DataGrid, and Tree)	Some list-based components have Spark equivalents (including List, HorizontalList, and TileList). You can use the UIFTETextField and MXFTETextInput classes in the other components if you do not use selection, editability, HTML links, or scrolling. Otherwise, you should embed a non-CFF version of the font to support these controls.
Label and Text	Use Spark equivalents such as Label, RichText, and RichEditableText. You can use UIFTETextField with the Label and Text controls if the text is not selectable or you do not use the `htmlText` property to specify the content of the controls. Otherwise, you should embed a non-CFF version of the font to support these controls.
TextInput and TextArea	Use the Spark equivalents. Otherwise, you should embed a non-CFF version of the font to support these controls.
RichTextEditor	There is no equivalent class. In this case, you should embed a non-CFF version of the font to support this control.

MX control	Description
ColorPicker	There is no equivalent class. In this case, you should embed a non-CFF version of the font to support this control. However, the ColorPicker control only uses the font to display a color value, so in some cases, using an embedded font might not be necessary.
ComboBox	Use the Spark equivalent. If your ComboBox's text does not need to be editable, you can use the MXFTETextInput class. Otherwise, you should embed a non-CFF version of the font to support this control.
DateField	If you do not use editability, then you can use the MXFTETextInput class. Otherwise, you should embed a non-CFF version of the font to support this control.
NumericStepper	If you do not use editability, then you can use the MXFTETextInput class. Otherwise, you should embed a non-CFF version of the font to support this control.

Embedding non-CFF versions of fonts for MX components

To use MX controls with embedded fonts, you can embed the non-CFF version of the font instead of changing the properties of the control to use CFF fonts. You should do this only if your MX control:

- Has no Spark equivalent
- Does not support using the UIFTETextField and MXFTETextInput classes for text rendering
- Must use selection, scrolling, or HTML text
- Is editable

If you embed a non-CFF version of a font in addition to a CFF version of the font in your application, your SWF file will be larger than if you embedded only a single version of the font. As a result, only do this when absolutely necessary.

If you compile an application with the `compatibility-version` compiler option set to 3.0.0, then the non-CFF version of the font is embedded automatically.

The following example embeds both a non-CFF version and a CFF version of the font so that the Spark and MX Labels use embedded fonts. The reason this is required is that the Label is selectable in this example.

```
<?xml version="1.0"?>
<!-- fonts/EmbedBoth.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myCFFFont;
        embedAsCFF: true;
     }

     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontNoCFF;
        advancedAntiAliasing: true;
        embedAsCFF: false;
     }

     .myCFFStyle {
        fontSize: 32;
        fontFamily: myCFFFont;
     }

     .myStyleNoCFF {
        fontSize: 32;
        fontFamily: myFontNoCFF;
     }
  </fx:Style>

  <s:Panel title="Using Two Different Embedded Fonts">
    <s:VGroup>
        <s:Label text="Spark Label"
            styleName="myCFFStyle"/>
        <mx:Label text="MX Label"
            styleName="myStyleNoCFF" selectable="true"/>
    </s:VGroup>
  </s:Panel>

</s:Application>
```

Note that when embedding a non-CFF font, you have the option of specifying the `advancedAntiAliasing` property. With CFF fonts, this property is ignored. The advanced anti-aliasing functionality is provided natively with FTE.

Troubleshooting fonts in applications

There are some techniques that you can use to successfully embed fonts into your applications.

Resolving compiler errors

The following table describes common compiler errors and their solutions:

Error	Solution
`Unable to resolve '`*`swf_file_name`*`' for transcoding`	Indicates that the font was not found by the compiler. Ensure that the path to the font is correct in the `@font-face` declaration or the `[Embed]` tag and that the path is accessible by the compiler.
`Font '`*`font_name`*`' with` *`style_description`* `not found`	Indicates that the `fontName` property used in the `[Embed]` statement might not match the name of the font. For fonts in SWF files, ensure that the spelling and word spacing of the font name in the list of available fonts in Flash is the same as the `fontName` property in your `[Embed]` statement and the `fontFamily` property that you use in your style definitions. This error can also mean that the font's style was not properly embedded in Flash. Open the FLA file and ensure that there is a text area with the font and style described, that the text is dynamic, and that you selected a character range for that text.

Resolving run-time errors

To determine if your fonts are embedded properly, you can use the isFontFaceEmbedded() method of the SystemManager, as described in "Detecting embedded fonts" on page 1581.

If you are using fonts embedded for Spark controls with MX controls, you should refer to "Embedding fonts with MX components" on page 1592.

To properly embed your fonts, try the following techniques:

- If one type of control is not correctly displaying its text, ensure that you are embedding the appropriate typeface. For example, the MX Button control's text labels require the bold typeface. If you do not embed the bold typeface, the MX Button control does not display the label's text.
- In your application, ensure that you set all properties for each font typeface in the `@font-face` declaration or `[Embed]` statement. To embed a bold typeface, you must set the `fontWeight` property to `bold`, as the following example shows:

  ```
  @font-face {
      src: url(../assets/MyriadWebProEmbed.ttf);
      fontFamily: "Myriad Web Pro";
      fontWeight: bold;
      embedAsCFF: true;
  }
  ```

  You also must set the `fontWeight` style property in your style definition:

  ```
  .myStyle2 {
      fontFamily: "Myriad Web Pro";
      fontWeight: bold;
      fontSize: 12pt;
  }
  ```

  If you use the `[Embed]` statement, you must set the `fontWeight` property to `bold` as the following example shows:

  ```
  [Embed(source="MyriadWebProEmbed.ttf", fontName="Myriad Web Pro",fontWeight="bold")]
  ```

- For fonts that are embedded in SWF files that you import, open the FLA file in Flash and ensure that all of the typefaces were added properly. Select each text area and do the following:
  - Check that the font name is correct. Ensure that the spelling and word spacing of the font name in the list of available fonts in Flash is the same as the `fontFamily` property in the `@font-face` declaration or the `fontName` property in your `[Embed]` statement. This value must also match the `fontFamily` property that you use in your style definitions.

    If you did not select an anti-aliasing option for the font in Flash 8 (for example, you chose Bitmap Text (no anti-alias)), you might need to change the value of the font name to a format that matches *fontName_fontSize*pt_st (for example, "Wingdings_8pt_st"). In the CSS for that bitmap font, be sure to set `fontAntiAliasType` to `normal`.

    To determine the exact font name exported by Flash (which you must match as the value of the `fontFamily` property in your application), open the SWF file in Flash and select Debug > Variables.
  - Check that the style is properly applied. For example, select the bold text area and check that the typeface really is bold.
  - Click the Embed button and ensure that the range of embedded characters includes the characters that you use in your application.
  - Check that each text area is set to Dynamic Text and not Static Text or Input Text. The type of text is indicated by the first drop-down box in the text's Properties tab.
  - Unless the SWF file was compiled with CFF, you must set the value of the embedAsCFF property to false for the imported font.
- For fonts in SWF files, ensure that you are using the latest SWF file that contains your fonts and was generated in Flash. Regenerate the SWF file in Flash if necessary.

- Ensure that the value of the `embedAsCFF` property is correct for the way that you are using a font. For Spark controls, set `embedAsCFF` to `true`. For fonts that are used by MX controls, try setting `embedAsCFF` to `false`. The following example sets the `embedAsCFF` property to `false` in the @font-face declaration:

```
@font-face {
    src: url(../assets/MyriadWebProEmbed.ttf);
    fontFamily: "Myriad Web Pro";
    fontWeight: bold;
    embedAsCFF: false;
}
```

About the font managers

***Note:** This is an advanced topic. Most developers do not need to adjust the font managers for their applications.*

Flex includes several font managers to handle embedded fonts. The font managers take embedded font definitions and draw each character in Flash Player. This process is known as *transcoding*. The font managers are Batik, JRE, AFE (Adobe Font Engine), and CFF, represented by the BatikFontManager, JREFontManager, AFEFontManager, and CFFFontManager classes, respectively.

The CFF font manager supports both TrueType and OpenType fonts. It also supports URL and system fonts. Use this manager for all CFF fonts.

The AFE font manager supports both TrueType and OpenType fonts. It also adds support for all non-CFF font embedding in Flex 4. The AFE font manager is the only font manager that you can use to transcode TrueType or OpenType fonts for non-CFF fonts. The fonts can only be referenced by a path to the font file, not by an OS-specific font name. If you embed an OpenType font, the compiler will use the AFE font manager to transcode the font because the other font managers do not support OpenType fonts, unless that OpenType font is a system font, in which case, the compiler will throw an error. None of the font managers can transcode OpenType fonts that are embedded as system fonts.

The Batik font manager transcodes only TrueType fonts, but does not support TrueType Collections (*.ttc). It does not transcode system fonts. If you specify the font location when you embed the font, the compiler will use the Batik font manager. In general, the Batik font manager provides smoother rendering and more accurate line metrics (which affect multiline text and line-length calculations) than the JRE font manager.

The JRE font manager transcodes TrueType system fonts, but the quality of output is generally not as good as the Batik font manager. If you install the font on your system, the compiler will use the JRE font manager because the Batik font manager does not support system fonts.

The following table shows which fonts are supported by which font managers:

	CFF	**Batik**	**AFE**	**JRE**
Font type	TrueType, OpenType	TrueType	TrueType, OpenType	TrueType
Method of embedding	URL, system	URL	URL	System

You determine which font managers the compiler can use in the flex-config.xml file. The default setting is to use all of them, as the following example shows:

```
<fonts>
    <managers>
        <manager-class>flash.fonts.JREFontManager</manager-class>
        <manager-class>flash.fonts.BatikFontManager</manager-class>
        <manager-class>flash.fonts.AFEFontManager</manager-class>
         <manager-class>flash.fonts.CFFFontManager</manager-class>
    </managers>
</fonts>
```

The preference of `<manager>` elements is in reverse order. This means that by default the CFF font manager is the preferred font manager; the compiler checks to see if a font can be transcoded using it first. If not, then the compiler checks to see whether the font can be transcoded using the AFE font manager and then the Batik font manager. Finally, if the other font managers fail, the compiler checks to see whether the JRE font manager can transcode the font.

Spark Skinning

You create Spark skins by either editing an existing skin class or creating a new skin class for a Spark component. For information about skinning MX components, see “Skinning MX components” on page 1655.

About Spark skins

In the Flex 4 skinning model, the skin controls all visual elements of a component, including layout. The new architecture gives developers greater control over what their components look like a structured and tool-friendly way. Previously, MX components that used the Halo theme for their skins defined their look and feel primarily through style properties.

Spark skins can contain multiple elements, such as graphic elements, text, images, and transitions. Skins support states, so that when the state of a component changes, the skin changes as well. Skin states integrate well with transitions so that you can apply effects to one or more parts of the skins without adding much code.

You typically write Spark skin classes in MXML. You do this with MXML graphics tags (or FXG components) to draw the graphic elements, and specify child components (or subcomponents) using MXML or ActionScript.

The base class for Flex 4 skins is the spark.components.supportClasses.Skin class. The default Spark skins are based on the SparkSkin class, which subclasses the Skin class.

In general, you should try to put all visual elements of a component in the skin class. This helps maintain a necessary separation between the model (the logic and declarative structure of the application) and the view (the appearance of the application). Properties that are used by skins (for example, the placement of the thumb in a slider control) should be defined in the component so that they can be shared by more than one skin.

Most skins use the BasicLayout layout scheme within the skin class. This type of layout uses constraints, which means that you specify the distances that each element is from another with properties such as `left`, `right`, `top`, and `bottom`. You can also specify absolute positions such as the x and y coordinates of each element in the skin.

When creating skins, you generally do not subclass existing skin classes. Instead, it is often easier to copy the source of an existing skin class and create another class from that. Use this method especially if you are going to reuse the skin for multiple instances of a component or multiple components. If you want to change the appearance of a single instance of a component, you can use MXML graphics syntax or apply styles inline.

When creating a Spark skin, you can use MXML, ActionScript, FXG, embedded images, or any combination of the above. You do not generally use run-time loaded assets such as images in custom skins.

More Help topics

Skinning Custom Skinnable Components (video)

Applying skins

You usually apply Spark skins to components by using CSS or MXML. With CSS, you use the `skinClass` style property to apply a skin to a component, as the following example shows:

```
s|Button {
    skinClass: ClassReference("com.mycompany.skins.MyButtonSkin");
}
```

When applying skins with MXML, you specify the name of the skin as the value of the component's `skinClass` property, as the following example shows:

```
<s:Button skinClass="com.mycompany.skins.MyButtonSkin" />
```

You can also apply a skin to a component in ActionScript. You call the `setStyle()` method on the target component and specify the value of the `skinClass` style property, as the following example shows:

```
myButton.setStyle("skinClass", Class(MyButtonSkin));
```

Anatomy of a skin class

Custom Spark skins are MXML files that define the logic, graphic elements, subcomponents, states, and other objects that make up a skin for a Spark component.

The structure of Spark skin classes is similar to other custom MXML components. They include the following elements:

- Skin root tag, or a subclass of Skin (required)
- Host component metadata (optional, but recommended)
- States declarations (required if defined on the host component)
- Skin parts (required if defined on the host component)
- Script block (optional)
- Graphic elements and other controls (optional)

In addition to these elements, Spark skins can contain MXML language tags such as Declarations and Library.

Root tags

Skin classes use the Skin class, or a subclass of Skin such as SparkSkin, as their root tag. The root tag contains the namespace declarations for all namespaces used in the skin class. The following commonly appears at the top of each skin class file:

```
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
```

You can set additional properties on the `<s:Skin>` tag, such as `minWidth` or `scaleX`. You can also set style properties such as `color` and `fontWeight`. In addition, you can specify values based on the state of the control on the root tag. For example, `color.down="0xFFFFFF"`. You cannot set the `includeIn` or `excludeFrom` properties on the root tag of the skin class.

If you create a custom theme for your application, or if you do not need support for global Spark styles in your custom skin class, you can use Skin rather than SparkSkin as your custom skin's root tag. The SparkSkin class adds support for the colorization styles (such as `chromeColor` and `symbolColor`) and supports excluding specific skin parts from colorization, or for specifying symbols to colorize.

Host components

Spark skin classes typically specify the host component on them. The host component is the component that uses the skin. By specifying the host component, Spark skins can gain a reference to the component instance that uses the skin by using the `hostComponent` property.

The following example from an MXML-based skin defines the Spark Button as the host component:

```
<fx:Metadata>
    [HostComponent("spark.components.Button")]
</fx:Metadata>
```

Adding the `[HostComponent]` metadata is optional, but it lets Flex perform compile-time checking for skin states and required skin parts. Without this metadata, no compile-time checking can be done.

You can specify a host component on an ActionScript-based skin, too. To do this, you declare a bindable public property called `hostComponent`. The `hostComponent` property must of the correct type. For example:

```
import spark.components.Button;
...
[Bindable]
public var hostComponent:Button;
```

The value of the `hostComponent` property is set by the component when the skin is loaded.

Skin states

Skin states are skin elements that are associated with component states. They are not the same as component states. For example, when a Button is down, the Button's skin displays elements that are associated with the `down` skin state. When the button is up, the button displays elements that are associated with the `up` skin state.

Skins must declare skin states that are defined on the host component. At runtime, the component sets the appropriate state on the skin. Skin states are referenced by using the dot-notation syntax, *`property.state`* (for example, `alpha.down` defines the value of the `alpha` property in the `down` state).

The following example shows the skin states for the Spark Button skin:

```
 <s:states>
    <s:State name="up" />
    <s:State name="over" />
    <s:State name="down" />
    <s:State name="disabled" />
</s:states>
```

Skin parts

Skin parts are components defined in the skin class and the host component. They often provide a way for a host component to push data into the skin. The component also uses skin parts to hook up behaviors.

Spark container skins include a content group that defines the group where the content children are pushed into and laid out in. This element has an ID of `contentGroup`. All skinnable containers have a `contentGroup`. The content group is a static skin part.

Layouts

Both the Skin and SparkSkin classes use BasicLayout as their default layout scheme. This is the equivalent of having the following defined in the skin class:

```
<s:layout>
    <s:BasicLayout/>
</s:layout>
```

The layout scheme is important when there is more than one graphical element or subcomponent used in the skin. BasicLayout relies on constraints and/or absolute positioning to determine where to place components.

Subcomponents

The Spark skin classes typically include graphic elements and other components that make up the appearance of the skin.

Script blocks

Optionally, the Spark skin class can include a Script block for skin-specific logic.

Most Spark skins have a special `<fx:Script>` block at the top of the skin class. This block typically defines style properties that the skin class respects, including the exclusions that the skin uses. The tag includes a special attribute, `fb:purpose="styling"`:

```
<fx:Script fb:purpose="styling">
```

This attribute is used by Flash Builder. When you create a copy of a skin class in Flash Builder, you can opt to make the skin styleable. If you choose to make it styleable, Flash Builder includes this section of the skin class. If you choose not to make the skin styleable, Flash Builder excludes this section.

Language tags

Like any MXML-based class, you can use the Library tag inside the root tag to declare repeatable element definitions. If you want to use non-visual objects in your skin class, you must wrap them in a Declarations tag.

Versions of included skin classes

While the new Spark skinning architecture makes creating your own skins easy, Flex 4 includes several sets of skins.

The following table describes the skinning packages that ship with Flex 4:

Package	Description
spark.skins.spark.*	Default skins for Spark components.
spark.skins.wireframe.*	A simplified theme for developing applications with a “prototype” look to them. To use wireframe skins, you can apply the wireframe theme or apply the skins on a per-component basis. For information about applying themes, see “Using themes” on page 1562.

Package	Description
mx.skins.halo.*	MX skins available for MX components that do not conform to the Spark skinning architecture. You can use these skins in your application instead of the Spark skins by overriding the styles, loading the Halo theme, or by setting the `compatibility-version` compiler option to 3.0.0 when compiling your application. For information about these skins, see "Skinning MX components" on page 1655.
mx.skins.spark.*	The default skins for MX components when using the default Spark theme. These skins are used by the MX components in Flex 4 applications. These skins give the MX components a similar appearance to the Spark components in Flex 4 applications.
mx.skins.wireframe.*	Wireframe skins for MX components.

Skins typically follow the naming convention *componentName*Skin.mxml. In the *ActionScript 3.0 Reference for the Adobe Flash Platform*, most skins have several versions. For example, there are four classes named ButtonSkin. The default skins for the Spark components are in the spark.skins.spark.* package.

Flex 4 also ships with several themes that use some of the skinning packages. For more information, see "About the included theme files" on page 1565.

Skinning contract

The *skinning contract* between a skin class and a component class defines the rules that each member must follow so that they can communicate with one another.

The skin class must declare skin states and define the appearance of skin parts. Skin classes also usually specify the host component, and sometimes bind to data defined on the host component.

The component class must identify skin states and skin parts with metadata. If the skin class binds to data on the host component, the host component must define that data.

The following table shows these rules of the skinning contract:

	Skin Class	Host Component

Host component	`<fx:Metadata>`
	`[HostComponent("spark.components.Button")]`
	`</fx:Metadata>`
		n/a

Skin states	`<s:states>`
	`<s:State name="up"/>`
	`</s:states>`
		`[SkinState("up")];`
		`public class Button {`
		`...`
		`}`

Skin parts	`<s:Button id="upButton"/>`	`[SkinPart(required="false")]`
		`public var upButton:Button;`

Data	`text="{hostComponent.title}"`	`[Bindable]`
		`public var title:String;`

The compiler validates the `[HostComponent]`, `[SkinPart]`, and `[SkinState]` metadata (as long as the `[HostComponent]` metadata is defined on the skin). This means that skin states and skin parts that are identified on the host component must be declared in the skin class.

For each [SkinPart] metadata in the host component, the compiler checks that a public variable or property exists in the skin. For each [SkinState] metadata in the host component, the compiler checks that a state exists in the skin. For skins with [HostComponent] metadata, the compiler tries to resolve the host component class, so it must be fully qualified.

After you have a valid contract between a component and its skin class, you can apply the skin to the component.

Accessing host components

Spark skins optionally specify a host component. This is not a reference to an instance of a component, but rather, to a component class. You define the host component by using a [HostComponent] metadata tag with the following syntax:

```
<fx:Metadata>
    [HostComponent(component_class)]
</fx:Metadata>
```

For example:

```
<fx:Metadata>
    [HostComponent("spark.components.Button")]
</fx:Metadata>
```

When a skin defines this metadata, Flex creates the typed property hostComponent on the skin class. You can then use this property to access members of the skin's host component instance from within the skin. For example, in a Button skin, you can access the Button's style properties or its data (such as the label).

You can access public properties of the skin's host component by using the strongly typed hostComponent property as a reference to the component.

```
<s:SolidColor color="{hostComponent.someColor as uint}" />
```

This only works with public properties that are declared directly on the host component. You cannot use this to access the host component's private or protected properties.

To access the values of style properties on a host component from within a skin, you are not required to specify the host component. You can use the getStyle() method as the following example shows:

```
<s:SolidColorStroke color="{getStyle('color')}" weight="1"/>
```

You can also access the root application's properties and methods by using the FlexGlobals.topLevelApplication property. For more information, see "Accessing application properties" on page 1623.

Defining skin states

Each skinnable component has a set of visual skin states. For example, when a Button is down, the Button's skin displays elements that are associated with the down skin state. When the button is up, the button displays elements that are associated with the up skin state.

To have a valid contract between a skinnable Spark component and its skin, you identify the skin states in the component. Then, define the state's appearance in the component's skin class.

Skin states are declared in the skin class and identify the different states that the component can assume visually. You can define how the visual appearance changes as the skin's state changes in the skin class.

Subclasses inherit the skin states of their parent. For example, the Button class defines the skin states up, down, over, and disabled. The ToggleButton class, which is a subclass of Button, declares the upAndSelected, overAndSelected, downAndSelected, and disabledAndSelected skin states, in addition to those states defined by the Button control.

Identifying skin states in a component

Part of the contract between a Spark skin and its host component is that the host component must identify the skin states that it supports. To identify a skin state in the component's class, you use the `[SkinState]` metadata tag. This tag has the following syntax:

```
[SkinState("state")]
```

You specify the metadata before the class definition. The following example defines four skin states for the Button control:

```
[SkinState("up")]
[SkinState("over")]
[SkinState("down")]
[SkinState("disabled")]
public class Button extends Component { .. }
```

Defining the skin states in the skin class

The Spark skinning contract requires that you declare supported skin states in the skin class. You can also optionally define the appearance of the skin state in the skin class. Even if you declare a skin state in the skin class, you are not required to define its appearance.

To define a skin state in a skin class:

1. Declare the skin state in a `<states>` tag.
2. Set the value of properties based on the state of the component. This step is optional, but if you don't define the skin state's appearance, then the skin does not change when the component enters that state.

To declare skin states in the skin class, you populate the top-level `<s:states>` tag with an array of State objects. Each State object corresponds to one skin state.

The following example defines four states supported by the Button skin class:

```
<s:Skin ...>
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    ...
</s:Skin>
```

After you declare the skin states in the skin class, you can then define the appearance of the skin states. To do this, you specify the values of the properties based on the skin state by using the dot-notation syntax (*`property_name.state_name`*). To set the value of the `weight` property of a SolidColorStroke object in the `over` state, you specify the value on the `weight.over` property, as the following example shows:

```
<s:SolidColorStroke color="0x000000" weight="1" weight.over="2"/>
```

You can also specify the stateful values of properties by using the `includeIn` and `excludeFrom` properties.

Most commonly, you specify values of style properties based on the state. Flex applies the style based on the current state of the component. The skin is notified when the component's `currentState` property changes, so the skin can update the appearance at the right time.

A common use of this in the Spark skins is to set the `alpha` property of a component when the component is in its `disabled` state. This is often set on the top-level tag in the skin class, as the following example from the ButtonSkin class shows:

```
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21"
    alpha.disabled="0.5">
```

Another example is when a Button label's `alpha` property changes based on other states. The skin sets the value of the labelDisplay's `alpha` property. When the button is in its `up` state, the label has an `alpha` of 1. When the button is in its `over` state, the label has an `alpha` of .25, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/StatesButtonExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Button label="Alpha Changes" skinClass="mySkins.MyAlphaButtonSkin"/>
</s:Application>
```

The skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\MyAlphaButtonSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">

    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
        alpha.up="1"
        alpha.down=".1"
        alpha.over=".25"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

You can set any property on a component based on the state. You are not limited to style properties. For example, you can change the label of a Button control in its skin based on its state, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ChangeLabelBasedOnStateExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
   <s:Button id="myButton" label="Basic Button" skinClass="mySkins.ChangeLabelBasedOnState"/>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\ChangeLabelBasedOnState.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
       <!-- Add this fill to make the "hit area" cover the entire button. -->
       <s:fill>
            <s:SolidColor color="0xFFFFFF"/>
       </s:fill>
    </s:Rect>

    <s:Label id="labelDisplay"
         text.up="UP" text.over="OVER" text.down="DOWN"
         horizontalCenter="0" verticalCenter="1"
         left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

If you identify a skin state on the component, but do not declare it on the skin class, Flex throws a compiler error if the `[HostComponent]` metadata is set. You are not required to define the appearance of the skin state in the skin class, but you are required to declare it in the skin class in this case.

If you try to set the value of a style property in a skin class on a state that is not identified by the host component's class, Flex throws a compiler error. This error checking helps to enforce the contract between the component and the skin.

In addition to conditionalizing the values of properties, you can also include or exclude graphic elements from the skin based on the component's state. To do this, you use the `includeIn` and `excludeFrom` properties to define which states a graphic element applies to.

The following example displays a partially transparent gray box when the Button control is in the `down` state:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/IncludeButtonExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Button label="Click Me In the Button Class" skinClass="mySkins.MyIncludeButtonSkin"
color="0xCCC333"/>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\MyIncludeButtonSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2">
    </s:Label>
    <!-- Highlight (down state only) -->
    <!-- Note that you might need to adjust the radiusX and radiusY properties. -->
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" includeIn="down">
        <s:fill>
            <s:SolidColor color="0x000000" alpha="0.25"/>
        </s:fill>
    </s:Rect>
</s:Skin>
```

When you exclude a graphic element with the `excludeFrom` property, Flex removes it from its parent's child list. The element can no longer be referenced.

If you do not specify either the `includeIn` or `excludeFrom` property for a graphic element in a skin class, the graphic element is used in all states of the control by default.

You can specify multiple states to be included or excluded by using a comma-separated list of states for the `includeIn` or `excludeFrom` properties. The following example excludes the Rect graphic element from the host component's `over` and `down` states:

```
<s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" excludeFrom="over, down">
```

The `excludeFrom` and `includeIn` properties can be set only in MXML. You cannot set the values of these properties in ActionScript.

All classes support the `includeIn` and `excludeFrom` properties when they are used in the skin class, as long as they are a visual child of a visual parent. This means that you cannot set these properties on the root tag of the skin file, nor can you set them on scalar properties. For example, the following shows a valid and an invalid use of the `includeIn` property:

```
<s:states>
    <s:State name="StateA" />
</s:states>

<!-- This is a valid use of the includeIn property: -->
<s:Button>
    <s:label.StateA>
        <fx:String>My Label</fx:String>
    </s:label.StateA>
</s:Button>

<!-- This is an invalid use of the includeIn property: -->
<s:Button>
    <s:label>
        <fx:String includeIn="StateA">My Label</fx:String>
    </s:label>
</s:Button>
```

If you exclude some graphic elements from a skin, the component sometimes resizes itself or its appearance fluctuates as it cycles through its states. For example, if you set the value of the `excludeFrom` property to `down` on a button skin's labelDisplay, the label disappears when the user clicks the button. As a result, the button resizes itself in the `down` state. This behavior is expected and is shown in the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/HideLabelOnDownExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Button id="myButton"
        skinClass="mySkins.HideLabelOnDownSkin"
        label="This Is A Long Button Label"/>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\HideLabelOnDownSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect
        left="0" right="0"
        top="0" bottom="0"
        width="69" height="20"
        radiusX="2" radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
         excludeFrom="down"
         horizontalCenter="0" verticalCenter="1"
         left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

The button resizes because its size is dynamically determined by the contents of the label by default. Excluding the label results in the label being removed from its parent.

To prevent the Button from resizing itself, you can set the `alpha.down` property to 0 or the `visible.down` property to `false` rather than excluding the component altogether. The label then disappears but the button does not resize itself because the label is not removed from its parent, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/HideLabelOnDownExample2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Button id="myButton"
        skinClass="mySkins.HideLabelOnDownSkin2"
        label="This Is A Long Button Label"/>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\HideLabelOnDownSkin2.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect
        left="0" right="0"
        top="0" bottom="0"
        width="69" height="20"
        radiusX="2" radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
         visible.down="false"
         horizontalCenter="0" verticalCenter="1"
         left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

Skin parts

Some components have multiple skin parts. For example, a NumericStepper contains an up button, a down button, and text. These parts are declared by the component. The skin defines their appearance. Parts defined by the component can be optional or required in the skin.

Skin parts are an important part of the skinning contract. They let the component instance interact with the skin and can be used to push data down into a skin or to define behaviors.

Skin parts do not have to be top-level tags in the Spark skin class. They can be anywhere in the MXML file.

To declare a skin part on a component, you use the `[SkinPart]` metadata. For example, the Button class defines a skin part for the label in the ButtonBase class:

```
[SkinPart(required="false")]
public var labelDisplaIDisplayText;
```

The ButtonSkin class defines a Label component as the labelDisplay part:

```
 <s:Label id="labelDisplay"
    textAlign="center"
    verticalAlign="middle"
     maxDisplayedLines="1"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
</s:Label>
```

In this example, the `labelDisplay` skin part is technically not required (`required="false"`), but without it, Flex would not draw a label on the Button control. The default value for the `required` property is `false`. The contract between the Button class and its skin dictates that when you set the `label` property on a button, the value is pushed down into the skin and modifies the value of the labelDisplay skin part's text, if the `labelDisplay` skin part exists.

Parts are identified by their `id` attributes. The `id` attribute of the skin part in the skin class must match the property name of the skin part in the host component. This helps enforce the contract between the skin and the host component. For example, if you have the following declaration in your component:

```
[SkinPart(required="true")]
public var textInput:TextInput;
```

The skin class sets the `id` attribute of a TextInput instance to "textInput", as the following example shows:

```
<s:TextInput id="textInput" ... />
```

There are two primary types of skin parts: static and dynamic. *Static parts* are instantiated automatically by the skin. There can be only one instance of a static part. For example, the VScrollBar class has four static parts: track, thumb, incrementButton, and descrementButton. The Button class has one static part: label.

Dynamic parts are instantiated when needed. There can be more than one instance of a dynamic part. Dynamic parts are defined in the Declarations block of a skin so that one or more instances of the part can be instantiated and used by the skin. The Slider class's DataTip, for example, is a dynamic skin part. In the VSliderSkin and HSliderSkin, the dataTip skin part is defined in a Declarations block.

In your host component, you specify the type for a dynamic skin part as an IFactory. For example, in the Slider class, the dataTip is defined as:

```
[SkinPart(required="false", type="mx.core.IDataRenderer")]
public var dataTip:IFactory;
```

At runtime, when a component skin is loaded, the SkinnableComponent base class finds all the skin parts, assigns parts that were found, and throws a runtime error if any required parts are not defined by the skin. Deferred parts are found (because they are defined properties of the skin), but have a value of null when the component is first instantiated.

Applying skin classes to a component

After you have defined a skin, you can apply it to a component in one of the following ways:

- CSS
- MXML
- ActionScript

To associate a skin with a component in CSS, you set the value of the `skinClass` style property in the style sheet. You can use either type or class selectors to apply a skin class to a component.

The following example applies the mySkins.NewPanelSkin class with all Panel containers by using a type selector:

```
@namespace s "library://ns.adobe.com/flex/spark";
s|Panel {
    skinClass:ClassReference("mySkins.NewPanelSkin");
}
```

The following example associates the mySkins.CustomButtonSkin class with all components that use the myButtonStyle class selector:

```
.myButtonStyle {
    skinClass:ClassReference("mySkins.CustomButtonSkin");
}
...
<s:Button label="Spark Button" className="myButtonStyle"/>
```

To associate a skin with a component in MXML, you set the value of the `skinClass` property inline. The following example applies the mySkins.NewPanelSkin class to this instance of the Panel container:

```
<s:Panel skinClass="mySkins.NewPanelSkin"/>
```

The advantage to applying skins with CSS is that with CSS you can use type and class selectors to apply the skin to all components of a particular type (such as all Buttons) or all classes that are in a particular class (such as all components with the style name “myButtonStyle”).

CSS supports inheritance. If you apply a skin class to a Button control in the Button type selector, the skin applies to all Button controls and subclasses of Button controls, such as the ToggleButton control. This can have unexpected results in the case of subcomponents. If you apply a new skin to all Label controls, components that have Label subcomponents will also use that skin. As a result, you should generally use class selectors for applying skins to basic components.

Setting a skin class in MXML lets you only apply the skin to the instance of the component.

You can also apply a skin class to a component in ActionScript. Call the `setStyle()` method on the component, and set the value of the `skinClass` style property to the skin class. You must cast the skin class as a Class in the `setStyle()` method.

The following example applies the custom skin class to the Button when you click the Button:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/SimpleLoadExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        import mySkins.*;

        private function changeSkinClickHandler():void {
            myButton.setStyle("skinClass", Class(MyButtonSkin));
        }
    </fx:Script>
    <s:Button id="myButton"
        label="Click Me"
        color="0xFFFFF"
        click="changeSkinClickHandler()"/>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\MyButtonSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2" 
radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
             horizontalCenter="0" verticalCenter="1"
             left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

Be sure to import the appropriate skin class package before you can use the class in your application.

As with CSS, you can use ActionScript to apply a skin class to all instances of a particular component type. Call the `setStyle()` method on the component's style declaration. The following example applies the custom Button skin to all buttons in the application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ASTypeSelectorLoadExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import mySkins.*;

        private function changeSkinClickHandler():void {
           styleManager.getStyleDeclaration("spark.components.Button").setStyle("skinClass",
Class(MyButtonSkin));
        }
    </fx:Script>
    <s:Button id="myButton1" label="Click Me" color="0xFFFFF"
        click="changeSkinClickHandler()"/>
    <s:Button id="myButton2" label="Click Me" color="0xFFFFF"
        click="changeSkinClickHandler()"/>
    <s:Button id="myButton3" label="Click Me" color="0xFFFFF"
        click="changeSkinClickHandler()"/>
</s:Application>
```

Documenting skin states and skin parts

The ASDoc utility supports documenting the `[SkinState]` and `[SkinPart]` metadata tags.

You can document the `[SkinState]` metadata tag by adding comments above the tag in the component's class file, as the following example shows:

```
/**
* Up state of the button.
*/
[SkinState("up")]
```

To document a `[SkinPart]` metadata tag, the ASDoc utility uses the description of the variable from the component's class file. You do not add a comment on the actual metadata tag, as the following example shows:

```
[SkinPart(required="false")]
/**
* A skin part that defines the  label of the button.
*/
public var labelDisplay:TextGraphicElement;
```

The ASDoc utility adds skin state documentation to the "Skin States" section of the class's documentation. It adds skin part documentation to the "Skin Parts" section of the class's documentation.

For more information about the ASDoc utility, see "ASDoc" on page 2241.

Creating skins from source files

When creating a custom Spark skin, the easiest way to get started is to use the source of an existing skin of a similar type as your base class. You can use the default Spark skins as a good starting point for creating skins, or the skins in the Wireframe theme. The Wireframe theme defines a set of simple skins that provide a "prototype" look to the application. They are lighter weight than the default skins but have less functionality.

The default Spark skins are more heavyweight than the Wireframe skins, but they let you maintain the default Spark look and feel. For example, if you are creating a custom Button skin, you can open the spark.skins.spark.ButtonSkin.mxml file. Save this file in a different location with a different name. You can then edit the contents of this file for your new custom skin.

The default skins have names that are similar to the component class name (for example, ToggleButton uses the ToggleButtonSkin class). If you are unsure of the skin class name is used by a component, look in the defaults.css file. This style sheet defines all the default skin classes for the components that have them.

In Flash Builder, you can create a new skin based on an existing skin in Design Mode. In the Properties view, click the Pencil icon for Skin and select Create New Copy of Skin. Flash Builder creates a file that implements a Skin class for the selected component. Flash Builder modifies the application file by specifying the new skin class as the `skinClass` property of the current component. Flash Builder also opens the generated Skin class file in Design mode of the MXML editor. For more information on using Flash Builder to design custom skins, see Generate and edit skins for Spark components.

When you copy a skin in Flash Builder, you can optionally include the style information to make the skin styleable. If you do not, then Flash Builder removes the style block that is tagged with `fx:purpose="styling"`. The skin will look similar, but will not respect styles such as `chromeColor`, `color`, and the exclusions.

Skins commonly contain graphic elements such as rectangles, lines, and ellipses, as well as visual effects such as blend modes, transforms, and filters. Using these graphic elements, sometimes referred to as MXML graphics, is simple and integrates well with most skins. For more information, see “MXML graphics” on page 1730.

Another way to create a spark skin is to generate its graphical elements from a tool that outputs FXG. FXG is a graphics interchange format that defines vector graphic elements. You can create a graphic in a tool such as Adobe Illustrator that can then export the instructions for drawing that graphic in the FXG format. You can either use the exported FXG as a component in your skin class, or you can convert the FXG syntax to use in your Spark skin class to define the appearance of your skins. Using FXG output as a custom component in your skin class is more efficient because the compiler optimizes FXG custom component source code. For more information about using FXG and converting it to MXML graphics, see “FXG” on page 1719.

Skinning Spark components

All Spark components and subcomponents that are subclasses of SkinnableComponent can be reskinned. Common tasks when reskinning Spark components include adding borders, drop shadows, and transitions to a skin class.

Setting minimum sizes of a skin

A common task when creating Spark skins is to set minimum sizes of the skin. This causes the Flex layout to not reduce the size of a component below a pre-defined width or height. You do this with the `minWidth` and `minHeight` properties on the skin class’s top-level element.

In general, you should not change the values of the minimum height and minimum width properties once they are set.

The following example creates a custom Button skin that has a minimum width of 100 pixels and a minimum height of 100 pixels.

```
<?xml version="1.0"?>
<!-- SparkSkinning/SquareButtonExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Button id="myButton" skinClass="mySkins.SquareButtonSkin" label="Click Me"/>

</s:Application>
```

The following is the custom skin class used for this example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/SquareButtonSkin.mxml -->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="100" minHeight="100"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <fx:Script>
        <![CDATA[
            static private const exclusions:Array = ["labelDisplay"];
            override public function get colorizeExclusions():Array {return exclusions;}
        ]]>
    </fx:Script>

    <s:states>
        <s:State name="up" />
        <s:State name="over" />
        <s:State name="down" />
        <s:State name="disabled" />
    </s:states>

    <!-- layer 1: shadow -->
    <s:Rect left="-1" right="-1" top="-1" bottom="-1" radiusX="2" radiusY="2">
        <s:fill>
            <s:LinearGradient rotation="90">
                    <s:GradientEntry color="0x000000"
                                  color.down="0xFFFFFF"
                                  alpha="0.01"
                                  alpha.down="0" />
                    <s:GradientEntry color="0x000000"
                                  color.down="0xFFFFFF"
                                  alpha="0.07"
                                  alpha.down="0.5" />
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <!-- layer 2: fill -->
    <s:Rect left="1" right="1" top="1" bottom="1" radiusX="2" radiusY="2">
        <s:fill>
```

```
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0xFFFFFF"
                               color.over="0xBBBDBD"
                               color.down="0xAAAAAA"
                               alpha="0.85" />
                <s:GradientEntry color="0xD8D8D8"
                               color.over="0x9FA0A1"
                               color.down="0x929496"
                               alpha="0.85" />
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <!-- layer 2: border -->
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
        <s:stroke>
            <s:LinearGradientStroke rotation="90" weight="1">
                <s:GradientEntry color="0x000000"
                               alpha="0.5625"
                               alpha.down="0.6375" />
                <s:GradientEntry color="0x000000"
                               alpha="0.75"
                               alpha.down="0.85" />
            </s:LinearGradientStroke>
        </s:stroke>
    </s:Rect>
    <s:Label id="labelDisplay"
             textAlign="center"
             verticalAlign="middle"
             lineBreak="toFit"
             horizontalCenter="0" verticalCenter="1"
             left="10" right="10" top="2" bottom="2">
    </s:Label>

</s:SparkSkin>
```

The default Button skin's minimum size is 21 pixels wide by 21 pixels high. These minimums are set on the root tag of the skin class, as the following example shows:

```
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21"
    alpha.disabled="0.5">
    ...
</s:SparkSkin>
```

You might notice that in the default button skin's class, though, the Rect border sets its width to a value that is larger than the minimum width on the skin:

```
 <s:Rect id="border" left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2">
```

The `width` property on the Rect border is used for buttons whose size is unset. The `minWidth` property on the SparkSkin is used when the button's width is variable, meaning that it hasn't been explicitly set. The previous example sets a default width and height of 69 and 20. However, it is pinned to (left=0, right=0, top=0, bottom=0) so that if the label is too large or the button is explicitly sized, then this rectangle will resize.

Accessing application properties

In addition to accessing the host component's properties, you can access properties of the application. This is useful if there are global settings that you want to access, or runtime information that is passed into the application that you want available in the skin class.

To access global variables in a custom skin, you use the `FlexGlobals.topLevelApplication` property. Using this property gives you access to all global variables, including variables that were passed into the application as `flashVars` variables.

The following example accesses a String that is a global variable on the application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/GlobalVariableAccessorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        public var myLabelString:String = "Hello World";
    </fx:Script>
    <s:Button skinClass="mySkins.GlobalVariableAccessorSkin"/>
</s:Application>
```

The following is the custom skin class for this example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\GlobalVariableAccessorSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>
    <fx:Script>
        import mx.core.FlexGlobals;

        [Bindable]
        private var localString:String = FlexGlobals.topLevelApplication.myLabelString;
    </fx:Script>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect
        left="0" right="0"
        top="0" bottom="0"
        width="69" height="20"
        radiusX="2" radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
        text="{localString}"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

Using style properties in custom skins

Within the skin, you can get the value of certain style properties on the host component. The component declares what styles can be set so that they can be set inline in MXML. For example, you can set the `chromeColor` property on all skins that extend the SparkSkin class. On other skins, such as container skins, you can set the `backgroundColor`.

The following example sets the color of the border inside the skin to the same value as the Button's `color` style property:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BorderColorButtonExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Button id="myButton"
        label="Basic Button"
        skinClass="mySkins.BorderColorButtonSkin"
        color="green"/>
</s:Application>
```

The following is the skin class for this example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\BorderColorButtonSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21"
    creationComplete="initSkin()">

    <fx:Script>
        private function initSkin():void {
            /* Note that because color can change, you could override the
                updateDisplayList() method and set it there so that the
                color is updated in the skin when it is updated on the
                host component. */
            outline.color = getStyle('color');
        }
    </fx:Script>

    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2" 
radiusY="2">
        <s:stroke>
            <s:SolidColorStroke id="outline" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

You can use a binding expression to set properties in the skin based on values in the host component. The following example binds the value of the stroke’s `color` property to the host component’s `color` style property:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/StyleWatcherExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Button id="myButton"
        label="Basic Button"
        skinClass="mySkins.StyleWatcherSkin"
        click="myButton.setStyle('color','red')"
        color="green"/>
</s:Application>
```

The following is the skin class for this example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\StyleWatcherSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">

    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <!-- Specify one state for each SkinState metadata in the host component's class -->
    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2"
radiusY="2">
        <s:stroke>
            <!-- Match the border color to the button label's color. -->
            <s:SolidColorStroke color="{getStyle('color')}" weight="1"/>
        </s:stroke>
    </s:Rect>

    <s:Label id="labelDisplay"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2">
    </s:Label>
</s:Skin>
```

This example sets the color of the stroke when the application starts. When you click the Button, the `color` property later changes while the application is running because style properties are bindable.

For custom components, you must declare the style metadata on the component itself so that inline MXML styling will work.

To expose a property as a style, you can also define a getter and setter for the property. You then override the `styleChanged()` method to dispatch the binding change event. In the application, you call the `setStyle()` method on the instance's `skin` property, which sets the value of the style property on the skin.

The following example exposes a custom style property called `borderColor` on the custom skin:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/StyleableBorderSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">

    <fx:Script>
        [Bindable("borderColorChange")]

        public function get borderColor():uint {
            return getStyle("borderColor");
        }

        public function set borderColor(value:uint):void {
            setStyle("borderColor", value);
        }
        override public function styleChanged(styleProp:String):void {
            super.styleChanged(styleProp);
            if (styleProp == "borderColor" || styleProp == null)
                dispatchEvent(new Event("borderColorChange"));
        }
    </fx:Script>
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>
    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>
    <s:Group id="contentGroup" left="0" right="0" top="0" bottom="0">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="{borderColor}" weight="1"/>
        </s:stroke>
    </s:Rect>
</s:Skin>
```

In the following application, you set the color of the border by using the ColorPicker control. This control uses the `setStyle()` method to change the color of the container's border.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/StyleableBorderExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" height="250" width="450">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <fx:Script>
        private function setSkinStyles(e:Event):void {
            myContainer.setStyle("borderColor",e.currentTarget.selectedColor);
        }
    </fx:Script>
    <s:SkinnableContainer id="myContainer"
        height="200" width="200"
        skinClass="mySkins.StyleableBorderSkin">
    </s:SkinnableContainer>

    <mx:ColorPicker id="myColorPicker" change="setSkinStyles(event)"/>
</s:Application>
```

Another way to expose style properties to the skin is to add code to the `updateDisplayList()` method to manually push style values into the skin's graphics. This is the more efficient method of pushing style properties to skins because it does not rely on binding. The following example overrides the `updateDisplayList()` method to push the background color's style property:

```
<fx:Script>
    override protected function updateDisplayList(unscaleWidth:Number,
unscaledHeight:Number):void {
        // Push style values into the graphics properties before calling
super.updateDisplayList
        backgroundFill.color = getStyle("backgroundColor");
        // Call super.updateDisplayList to do the rest of the work
        super.updateDisplayList(unscaledWidth, unscaledHeight);
    }
</fx:Script>
<s:Rect left="0" right="0" top="0" bottom="0">
    <s:SolidColor id="backgroundFill" />
</s:Rect>
```

Using events in custom skins

In general, you should use states to react to user interaction. If a skin part is removed from display list whenever the mouse hovers over a control, then that part cannot receive mouseDown events. Some skins, such as the Button control's skin, block user interaction to the skin so that the underlying control only can dispatch events. This means that you cannot trigger events defined on InteractiveObject on the skin. These include user interaction events such as `mouseDown`, `mouseOut`, and `click`.

You can trigger events defined on UIComponent on the skin, however. These events include `creationComplete` and other lifecycle events.

A Button control's default skin defines several Rect elements and a Label control. These simple classes do not support any events. To add lifecycle event listeners, you can wrap a simple element or series of elements in a Group tag. This lets you register lifecycle events that are defined on UIComponent.

The following example shows a Group inside the skin that triggers a `creationComplete` event:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/LifecycleEventExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="200" width="200">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:Group width="150" height="100">
        <s:Button label="Click It or Ticket" skinClass="mySkins.LifecycleEventSkin"/>
    </s:Group>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\LifecycleEventSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect id="buttonBorder" width="100%" height="20" radiusX="2" radiusY="2">
        <s:stroke>
            <mx:SolidColorStroke color="0x000000" weight="1" weight.over="2"/>
        </s:stroke>
    </s:Rect>

    <s:Group creationComplete="trace('creationComplete')">
        <s:Label id="labelDisplay"
            horizontalCenter="0" verticalCenter="1"
            left="10" right="10" top="6" bottom="2">
        </s:Label>
    </s:Group>
</s:Skin>
```

Skinning Spark containers

Spark defines two types of containers:

- Skinnable containers
- Non-skinnable containers

Both types of containers are in the spark.components.* package. Most Spark containers are skinnable. For example, the SkinnableContainer, Panel, and TitleWindow containers are skinnable. Skinnable containers subclass the SkinnableContainer class.

Spark groups are examples of containers that are not skinnable. They provide a lightweight mechanism that you use to perform layout. However, Spark groups do not support skinning to ensure that they add minimal overhead to your application. To modify the visual appearance of a Spark group, you can use the corresponding Spark container, which is skinnable. By contrast, all MX containers are skinnable.

Skinnable container skins are just like other component skins. They define a group that defines where the content children are laid out. This element has an ID of `contentGroup`. All skinnable containers have a content group. All visual children of a container are pushed into the content group and laid out using that group's layout rules.

For a list of which containers are skinnable and/or scrollable, see "About Spark containers" on page 417.

Sizing Spark container skins

A common reason to skin a Spark container is to add a graphic element, such as a rectangle, to be the background of the container. You typically set its size by using the constraint properties or the percent size properties by using one of the following methods:

- Constraints (setting top=0, bottom=0, left=0, right=0)
- Percent sizes (setting height=100%, width=100%)

When the components are first drawn, constraint properties take precedence (so if you set them, then the dimensional properties are ignored). However, if you set the value of the constraint properties, but then later explicitly set the value of the percent size properties at runtime (for example, by setting the `percentHeight` property to 100), Flex honors the new settings.

Using constraint or percent size properties depends on the use case. In general, a skin resizes when the host component resizes. The choice of percent or constraint sizing is based on the resizing scenario. For example, if you want a label element to be always half as wide as the skin, then set the `width` property to 50%. If you want a label element to be always inset by 5 pixels, set the `left` and `right` properties to 5.

The resizable skin elements in the default Spark skins usually set the `left` and `right` properties. To specify their default size, they also set the `width` property. As a result, if the component size is not set in the application, the element's width sets the default size of the skin and the component.

The following example from the ButtonSkin.mxml class shows that the default size of the Button control is the default size of its skin. This is calculated from the rectangle in the skin, and is 69 pixels wide and 20 pixels high:

```
<s:Rect left="0" right="0" top="0" bottom="0" width="69" height="20" radiusX="2" radiusY="2">
    <s:stroke>
        <s:LinearGradientStroke rotation="90" weight="1">
            <s:GradientEntry color="0x000000" alpha="0.5625" alpha.down="0.6375" />
            <s:GradientEntry color="0x000000" alpha="0.75" alpha.down="0.85" />
        </s:LinearGradientStroke>
    </s:stroke>
</s:Rect>
```

Another technique for designing resizable skins is to specify a minimum size for the skin. For more information, see "Setting minimum sizes of a skin" on page 1620.

Adding borders to Spark containers

One common use of a custom Spark skin is to add a borders to a container. Borders can be simple boxes around the perimeter of a container, or they can define drop shadows, line styles, corner radii, and other properties of a border.

To add a simple rectangular border to a container's skin, you add a Rect object. You can set the height and width of the Rect to 100% so that the skin sizes itself to the size of the container automatically, or you can set the offsets to 0.

The following example adds a simple 1-point, black line as a border around the container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/SimpleContainerBorderExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableContainer id="myContainer" height="200" width="200"
skinClass="mySkins.ContainerBorderSkin">
        <s:Button label="Click Me"/>
    </s:SkinnableContainer>
</s:Application>
```

The `contentGroup` container has left, right, top, and bottom offsets of 1. This lets the border display outside the content instead of sitting directly on top of the outside edges of container children. The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ContainerBorderSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>

    <s:Group id="contentGroup" left="1" right="1" top="1" bottom="1">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>
</s:Skin>
```

The container's skin sets its constraint properties (`top`, `bottom`, `left`, and `right`) to 0, so the Rect object is set to the same size as the container.

To create a border with rounded corners in a skin, you set the values of the `radiusX` and `radiusY` properties of the Rect object. These properties define the number of pixels for the corners on the x and y axes of the Rect. The following example sets the `radiusX` and `radiusY` properties to 10 to round the corners of the container's skin:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/RoundedContainerBorderExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableContainer id="myContainer"
        height="200" width="200"
        skinClass="mySkins.ContainerRoundedBorderSkin">
        <s:Button label="Click Me"/>
    </s:SkinnableContainer>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ContainerRoundedBorderSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>

    <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
    <s:Rect left="0" right="0" top="0" bottom="0" radiusX="10" radiusY="10">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>
</s:Skin>
```

You can add a border to a container by using the BorderContainer class. The BorderContainer class is a subclass of the SkinnableContainer class. Because you use CSS styles and class properties to control the appearance of the BorderContainer class, you typically do not create a custom skin for it.

The following example shows that the BorderContainer class takes the `cornerRadius`, `borderColor`, `borderVisible`, and `borderAlpha` style properties in addition to layout properties, which let you define a border similar to that shown in the previous example without skinning:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BorderContainerExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:BorderContainer width="200" height="200"
        borderColor="0x000000"
        borderAlpha="1"
        cornerRadius="10"
        borderWeight="1">
        <s:layout>
            <s:HorizontalLayout
                paddingLeft="15" paddingRight="15"
                paddingTop="15" paddingBottom="15"/>
        </s:layout>
        <s:Button label="Click Me"/>
    </s:BorderContainer>

</s:Application>
```

Because the border-related properties are styles, you can use CSS to define a consistent border across all instances of the BorderContainer class; for example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BorderContainerStyleExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";

        s|BorderContainer {
            borderColor:#000000;
            borderAlpha:1;
            cornerRadius:10;
            borderWeight:1;
        }
    </fx:Style>
    <s:BorderContainer width="200" height="200">
        <s:layout>
            <s:HorizontalLayout
                paddingLeft="15" paddingRight="15"
                paddingTop="15" paddingBottom="15"/>
        </s:layout>
        <s:Button label="Click Me"/>
    </s:BorderContainer>

</s:Application>
```

To add a border to a single instance of a container or group, you are not required to create a custom skin for the container. You can instead add a Rect graphic in the application itself. This is a process known as *composition*, where you are adding graphical elements to the application that change the appearance of controls in the application. Sometimes skinning gives a better abstraction because it distinguishes between visual children and content children.

For example, you can define MXML graphic tags fragment that draws the graphic elements that make up the border. These graphic elements are typically a stroke or fill (or both).

The following example draws a border around a VGroup that contains Button controls without using a custom skin class:

```
<?xml version="1.0"?>
<!-- SparkSkinning/SimpleBorderExampleNoSkin.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:Group>
        <!-- border/background graphics -->
        <s:Rect width="100%" height="100%">
               <s:stroke>
                   <s:SolidColorStroke color="0x000000" weight="2"/>
               </s:stroke>
        </s:Rect>
        <!-- content of container -->
        <s:VGroup left="10" top="10" right="10" bottom="10">
            <s:Button label="Click Me"/>
            <s:Button label="Click Me"/>
        </s:VGroup>
    </s:Group>

</s:Application>
```

Adding drop shadows to Spark containers

To add a drop shadow to a Spark container, you use the DropShadowFilter class. You add the filter to one of the graphic elements defined on that container's skin class. The DropShadowFilter class supports the `alpha`, `angle`, `distance`, and other properties that let you customize the appearance of the drop shadow. In addition, you add a fill to the graphic element. The fill defines the color of the drop shadow. These settings are all in the custom skin class and not in the main application.

The following example defines a drop shadow on the container's skin:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/DropShadowBorderExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableContainer id="myContainer"
        height="200" width="200"
        skinClass="mySkins.DropShadowBorderSkin">
        <s:Button label="Click Me"/>
    </s:SkinnableContainer>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/DropShadowBorderSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal"/>
        <s:State name="disabled"/>
    </s:states>
    <!-- drop shadow -->
    <s:Rect left="0" top="0" right="0" bottom="0">
        <s:filters>
            <s:DropShadowFilter
                blurX="20" blurY="20"
                alpha="0.32"
                distance="11"
                angle="90"
                knockout="true"/>
        </s:filters>
        <s:fill>
            <s:SolidColor color="0x000000"/>
        </s:fill>
    </s:Rect>

    <!-- layer 1: border -->
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" alpha="0.50" weight="1"/>
        </s:stroke>
    </s:Rect>
    <s:Group id="contentGroup" left="0" right="0" top="0" bottom="0">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
</s:Skin>
```

In this example, the drop shadow is the first element on the skin. You can use the `depth` property to change the ordering of the elements. This property determines the order in which items inside of containers are rendered. Spark containers order their items based on their `depth` property, with the lowest depth in the back, and the higher in the front. Items with the same `depth` value appear in the order they are added to the container.

A more efficient alternative to the DropShadowFilter class is to use the `dropShadowEnabled` property to add a drop shadow. For more information, see “Using the dropShadowEnabled property” on page 1558

You can also add a drop shadow to any individual container in the application itself through composition. The downside to doing it this way is that the drop shadow is applied to a single instance of the container, and not all containers. It also removes the benefits of abstraction that skins provide.

Adding padding to Spark containers

Another common use of Spark skins is modify the padding of Spark containers.

To add padding, you can set the offset properties of the container's content group. These properties are `top`, `bottom`, `left`, and `right`.

The following example sets the offsets of the contentGroup to 10 so that the contents of the container are padded by 10 pixels in all directions:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/PaddedContainerBorderExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:SkinnableContainer id="myContainer"
        height="200" width="200"
        skinClass="mySkins.ContainerPaddedBorderSkin"
    >
        <s:Button label="Click Me"/>
    </s:SkinnableContainer>
</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ContainerPaddedBorderSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>

    <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>
</s:Skin>
```

You can also add padding to the layout object. For example, the VerticalLayout object takes `paddingLeft`, `paddingRight`, `paddingBottom`, and `paddingTop` properties.

Specifying the layout in the skin changes the default layout for this container. However, you can set it on the container instance itself to override the default layout that you set in the skin.

More Help topics

"Using containers" on page 328

Adding scroll bars to Spark containers

Spark containers do not have scroll bars by default. To add scroll bars to a Spark container, use the Scroller class.

To add scroll bars to a container's skin class, you wrap the content group or other element in a Scroller tag. The child of a Scroller tag must implement the IViewport interface. This interface is implemented by Group and DataGroup, and some components such as the RichEditableText control.

To ensure that scroll bars appear, do not set explicit sizes on the group. If you set the size of the group explicitly, then that size becomes the size of the Group and no scroll bars will be shown.

The following example defines the size of the application, and sizes the container to a percent of the overall height and width. The scroller is constrained to the size of the viewport (in this case, the content group). Because the image is larger than the size of its parent, Flex displays scroll bars.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ScrollbarContainerExample.mxml -->
    <s:Application
        xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:mx="library://ns.adobe.com/flex/mx"
        xmlns:s="library://ns.adobe.com/flex/spark" height="200" width="200">

        <s:SkinnableContainer id="myContainer"
            height="50%" width="50%"
            skinClass="mySkins.ScrollBarContainerSkin">
            <s:Image source="@Embed(source='../assets/myImage.jpg')"/>
        </s:SkinnableContainer>

 </s:Application>
```

This example uses the following custom skin class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ScrollBarContainerSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Scroller height="100%" width="100%">
        <s:Group id="contentGroup">
            <s:layout>
                <s:BasicLayout/>
            </s:layout>
        </s:Group>
    </s:Scroller>
</s:Skin>
```

The Scroller layout does not support the left/right/top/bottom constraints on the content group of the container. And in general, you should not explicitly set the `height` and `width` properties of the Scroller's content group. Instead, you should let the content group size naturally.

You can use the `minViewportInset` property on the Scroller class to inset the viewport relative to its Scroller along the edges where a scroll bar does not already keep the viewport away from the edge. For example, if you set the `minViewportInset` property to 10, then the right edge of the viewport would be 10 pixels from the right edge of its scroller, unless the vertical scroll bar was visible. When a scroll bar is visible, the viewport and the scroll bar would not have any space between them.

More Help topics

"Scrolling Spark containers" on page 338

Adding background fills and images to Spark containers

To add background fills and background images to a Spark container's skin, you add a Rect or other graphic element and add the fill to that element.

To add a color fill, you add a graphic element that is a subclass of FilledElement and define a child fill tag. Subclasses of FilledElement include Rect, Ellipse, and Path. The fill must implement the IFill interface, which includes the BitmapFill, LinearGradient, RadialGradient, or SolidColor classes.

The following example defines three containers with three different fills:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BackgroundFillExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="700">

    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:SkinnableContainer id="myContainer"
        height="200" width="200"
        skinClass="mySkins.BackgroundFillSkin">
        <s:Button label="Basic Fill"/>
    </s:SkinnableContainer>
    <s:SkinnableContainer id="myContainer2"
        height="200" width="200"
        skinClass="mySkins.RadialBackgroundFillSkin">
        <s:Button label="Radial Gradient Fill"/>
    </s:SkinnableContainer>
    <s:SkinnableContainer id="myContainer3"
        height="200" width="200"
        skinClass="mySkins.LinearBackgroundFillSkin">
        <s:Button label="Linear Gradient Fill"/>
    </s:SkinnableContainer>
</s:Application>
```

The example uses the following custom skin class for the radial gradient fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/RadialBackgroundFillSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>
    <!-- layer 1: border -->
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0" alpha="0.50" weight="1" />
        </s:stroke>
    </s:Rect>
    <!-- background fill -->
    <s:Rect id="background" left="1" top="1" right="1" bottom="1">
        <s:fill>
            <s:RadialGradient>
                <s:entries>
                    <s:GradientEntry color="0xFFAABB" ratio="0" alpha="1"/>
                    <s:GradientEntry color="0xFFCCDD" ratio=".33" alpha="1"/>
                    <s:GradientEntry color="0xFFEEFF" ratio=".66" alpha="1"/>
                </s:entries>
            </s:RadialGradient>
        </s:fill>
    </s:Rect>

    <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
</s:Skin>
```

The example uses the following custom skin class for the basic background fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/BackgroundFillSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>
    <!-- layer 1: border -->
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0" alpha="0.50" weight="1" />
        </s:stroke>
    </s:Rect>
    <!-- background fill -->
    <s:Rect id="background" left="1" top="1" right="1" bottom="1">
        <s:fill>
            <s:SolidColor id="bgFill" color="0xFF0000"/>
        </s:fill>
    </s:Rect>

    <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
</s:Skin>
```

The example uses the following custom skin class for the linear background fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/LinearBackgroundFillSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>
    <!-- layer 1: border -->
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <mx:SolidColorStroke color="0" alpha="0.50" weight="1"/>
        </s:stroke>
    </s:Rect>
    <!-- background fill -->
    <s:Rect id="background" left="1" top="1" right="1" bottom="1">
        <s:fill>
            <s:LinearGradient>
                <s:entries>
                    <s:GradientEntry color="0xFFAABB" ratio="0" alpha="1"/>
                    <s:GradientEntry color="0xFFCCDD" ratio=".33" alpha="1"/>
                    <s:GradientEntry color="0xFFFFFF" ratio=".66" alpha="1"/>
                </s:entries>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
</s:Skin>
```

You can also add a background fill to a content group, although the content group often defines an offset or padding properties. These constraints are applied to the background in addition to the content. As a result, you typically add the background fill to a graphic element that is set to the entire height and width of the container.

To add a background image, you can use the BitmapFill class or a BitmapImage fill. The following example adds a background with a BitmapFill object:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/BackgroundFillExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:SkinnableContainer id="myContainer"
        height="256" width="206"
        skinClass="mySkins.BitmapFillBackgroundSkin"
    >
        <s:Button label="Bitmap Fill"/>
    </s:SkinnableContainer>
</s:Application>
```

This example uses the following custom skin class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/BitmapFillBackgroundSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.SkinnableContainer")]
    </fx:Metadata>

    <s:states>
        <s:State name="normal" />
        <s:State name="disabled" />
    </s:states>
    <!-- layer 1: border -->
    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0" alpha="0.50" weight="3" />
        </s:stroke>
    </s:Rect>
    <!-- background fill -->
    <s:Rect id="background" left="3" top="3" right="3" bottom="3" alpha=".25">
        <s:fill>
            <s:BitmapFill source="@Embed(source='../../assets/myImage.jpg')"/>
        </s:fill>
    </s:Rect>

    <s:Group id="contentGroup" left="10" right="10" top="10" bottom="10">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
</s:Skin>
```

In this example, the container is perfectly sized to fit the bitmap background plus the width of the border. In many cases, though, you will have to set properties on the fill to avoid tiling or apply clipping. For more information, see the BitmapFill class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The content group of the container's skin is typically the last element defined in the skin class. This is because the skin's layers are applied in the order in which they appear in the skin class. When a skin class has a background fill in a skin class, you add the content layer after it so that the content is not covered by the fill.

You can use the `depth` property, which gives you more control over the z-index. This lets you rearrange the appearance of the skin's elements so that it is not determined by the child order.

Swapping skins

You cannot unload or remove a skin class from its host component without specifying another skin to take its place. Otherwise, the component would have no definition of its visual parts (and would therefore break the skinning contract).

To swap a skin, you set the component's `skinClass` property to another skin class.

The following example applies a custom skin to the button when the application starts up. It then toggles the custom skin with the Button control's default skin class when you click the button:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/SimpleUnloadExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        import spark.skins.spark.ButtonSkin;
        import mySkins.MyButtonSkin;
        import flash.utils.*;

        private function toggleSkin():void {
            if (getQualifiedClassName(myButton.getStyle("skinClass")) ==
getQualifiedClassName(MyButtonSkin)) {
                // Load the default skin class.
                myButton.setStyle("skinClass", Class(ButtonSkin));
            } else {
                // Load the custom skin class.
                myButton.setStyle("skinClass", Class(MyButtonSkin));
            }
        }
    </fx:Script>
    <s:Button id="myButton"
        skinClass="mySkins.MyButtonSkin"
        label="Toggle Skin"
        color="0x0099FF"
        click="toggleSkin()"/>
</s:Application>
```

Transitions with Spark skins

You can use transitions to add visual appeal to your Spark skins. Transitions are triggered off of state changes, which are explicitly supported in Spark skins. All the visuals for a transition should be defined in the skin class and not on the component.

You can use transitions in Spark skins in the same way you would use them in your application. You add a `<s:transitions>` tag as a child tag of the skin's root tag. You then define the transitions and which states they apply to with the `toState` and `fromState` on the `<s:Transition>` child tags. Because the skin is notified of state changes from the host component, you do not have to add any logic to support state changes to the skin.

Typically, you specify the values for transitions in states and not in the effect itself.

The following example uses the Resize transition to grow and shrink the Button control's label and border. The transitions are triggered when the user rolls the pointer over the Button control and when the user rolls the pointer off of the Button control.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ButtonTransitionExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="200" width="200">

    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:Button label="Click Me"
        skinClass="mySkins.ButtonTransitionSkin"/>

</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning\mySkins\ButtonTransitionSkin.mxml -->
<s:Skin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21">
    <s:transitions>
        <s:Transition fromState="up" toState="over">
            <s:Resize target="{buttonBorder}"/>
        </s:Transition>
        <s:Transition fromState="over" toState="up">
            <s:Resize target="{buttonBorder}"/>
        </s:Transition>
    </s:transitions>
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <s:states>
        <s:State name="up"/>
        <s:State name="over"/>
        <s:State name="down"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Rect id="buttonBorder" width="69" width.over="79" height="20" radiusX="2" radiusY="2">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <!-- layer 8: text -->
    <s:Label id="labelDisplay"
        horizontalCenter="0" verticalCenter="1">
    </s:Label>
</s:Skin>
```

You typically declare the state values in the object or skin, and the effects use these values. The previous example defines the `width.over` property on the Rect rather than using the `widthBy` property of the effect to set the target width.

When using transitions inside Spark skins, you might have to assign IDs to elements that do not by default have IDs. This is because transitions require a target when you define them, and without an ID, you cannot target a specific element.

As with standard Flex controls, you can use transitions with multiple effects that are playing in parallel on a Spark skin. The following example resizes and changes the color of the button's label when the user moves their mouse over or out of the Button:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ButtonParallelTransitionExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="200" width="200">

    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:Button label="Click It or Ticket"
        skinClass="mySkins.ButtonParallelTransitionSkin"/>

</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ButtonParallelTransitionSkin.mxml -->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21"
    alpha.disabled="0.5">
    <s:transitions>
        <s:Transition>
            <s:Parallel target="{labelDisplay}">
                    <s:Animate>
                        <s:SimpleMotionPath property="fontSize"/>
                    </s:Animate>
                    <s:AnimateColor/>
            </s:Parallel>
        </s:Transition>
    </s:transitions>
    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <fx:Script>
        static private const exclusions:Array = ["labelDisplay"];
        override public function get colorizeExclusions():Array {return exclusions;}
    </fx:Script>

    <s:states>
```

```
        <s:State name="up" />
        <s:State name="over" />
        <s:State name="down" />
        <s:State name="disabled" />
    </s:states>

    <!--  The following values are negative because they define a border or drop shadow.
           The negative values separate them from the Button's bounds. -->
    <s:Rect left="-1" right="-1" top="-1" bottom="-1" radiusX="2" radiusY="2">
        <s:fill>
            <s:LinearGradient rotation="90">
                    <s:GradientEntry color="0x000000"
                                   color.down="0xFFFFFF"
                                   alpha="0.01"
                                   alpha.down="0" />
                    <s:GradientEntry color="0x000000"
                                   color.down="0xFFFFFF"
                                   alpha="0.07"
                                   alpha.down="0.5" />
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <s:Rect left="1" right="1" top="1" bottom="1" radiusX="2" radiusY="2">
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0xFFFFFF"
                               color.over="0xBBBDBD"
                               color.down="0xAAAAAA"
                               alpha="0.85" />
                <s:GradientEntry color="0xD8D8D8"
                               color.over="0x9FA0A1"
                               color.down="0x929496"
                               alpha="0.85" />
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <s:Rect left="1" right="1" bottom="1" height="9" radiusX="2" radiusY="2">
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0x000000" alpha="0.0099" />
                <s:GradientEntry color="0x000000" alpha="0.0627" />
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <s:Rect left="1" right="1" top="1" height="9" radiusX="2" radiusY="2">
        <s:fill>
            <s:SolidColor color="0xFFFFFF"
                        alpha="0.33"
                        alpha.over="0.22"
                        alpha.down="0.12" />
        </s:fill>
    </s:Rect>

    <s:Rect left="1" right="1" top="1" bottom="1"
```

```
            radiusX="2" radiusY="2" excludeFrom="down">
            <s:stroke>
                <s:LinearGradientStroke rotation="90" weight="1">
                    <s:GradientEntry color="0xFFFFFF" alpha.over="0.22" />
                    <s:GradientEntry color="0xD8D8D8" alpha.over="0.22" />
                </s:LinearGradientStroke>
            </s:stroke>
        </s:Rect>

        <s:Rect left="1" top="1" bottom="1" width="1" includeIn="down">
            <s:fill>
                <s:SolidColor color="0x000000" alpha="0.07" />
            </s:fill>
        </s:Rect>
        <s:Rect right="1" top="1" bottom="1" width="1" includeIn="down">
            <s:fill>
                <s:SolidColor color="0x000000" alpha="0.07" />
            </s:fill>
        </s:Rect>
        <s:Rect left="2" top="1" right="2" height="1" includeIn="down">
            <s:fill>
                <s:SolidColor color="0x000000" alpha="0.25" />
            </s:fill>
        </s:Rect>
        <s:Rect left="1" top="2" right="1" height="1" includeIn="down">
            <s:fill>
                <s:SolidColor color="0x000000" alpha="0.09" />
            </s:fill>
        </s:Rect>

        <s:Rect left="0" right="0" top="0" bottom="0"
            width="69" height="20"
            radiusX="2" radiusY="2">
            <s:stroke>
                <s:LinearGradientStroke rotation="90" weight="1">
                    <s:GradientEntry color="0x000000"
                               alpha="0.5625"
                               alpha.down="0.6375" />
                    <s:GradientEntry color="0x000000"
                               alpha="0.75"
                               alpha.down="0.85" />
                </s:LinearGradientStroke>
            </s:stroke>
        </s:Rect>
        <s:Label id="labelDisplay"
                 fontSize="{hostComponent.getStyle('fontSize')}"
                 fontSize.over="{hostComponent.getStyle('fontSize') + 4}"
                 color.over="0xFF0000"
                 textAlign="center"
                 verticalAlign="middle"
                 lineBreak="toFit"
                 maxDisplayedLines="1"
                 horizontalCenter="0" verticalCenter="1"
                 left="10" right="10" top="2" bottom="2">
        </s:Label>
    </s:SparkSkin>
```

The previous example uses the SparkSkin class rather than the Skin class as its root because it defines the `colorExclusions()` getter.

You can use transitions and states to effectively add and remove elements from the skin. For example, if you want an image to appear on a Button control during a mouse over, you can change the value of a property `height` from 0 to a new value in the `over` state.

The following example displays an image of a butterfly when you move the mouse over the Button control, and removes the image when you move the mouse out:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/ButterflySkinExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    backgroundColor="0x999999">

    <s:Button id="myButton"
        fontWeight="bold"
        color="0xFFFFFF"
        label="Bug of the Day"
        skinClass="mySkins.ButterflySkin"/>

</s:Application>
```

The custom skin class for this example is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/ButterflySkin.mxml -->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="21" minHeight="21" alpha.disabled="0.5">

    <!-- host component -->
    <fx:Metadata>
        <![CDATA[
        [HostComponent("spark.components.Button")]
        ]]>
    </fx:Metadata>

    <s:transitions>
        <mx:Transition fromState="up" toState="over">
            <mx:Parallel duration="350">
                <s:Resize target="{ myImage }"/>
                <s:Fade targets="{ [myImage,labelDisplay] }"/>
            </mx:Parallel>
        </mx:Transition>
        <mx:Transition fromState="over" toState="up">
            <mx:Parallel duration="200">
                <s:Resize target="{ myImage }"/>
                <s:Fade targets="{ [myImage,labelDisplay] }"/>
            </mx:Parallel>
        </mx:Transition>
    </s:transitions>
    <fx:Script>
        <![CDATA[
            static private const exclusions:Array = ["labelDisplay"];

            override public function get colorizeExclusions():Array {return exclusions;}
            override protected function initializationComplete():void  {
                useChromeColor = true;
                super.initializationComplete();
            }

            override protected function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number) : void  {
                var cr:Number = getStyle("cornerRadius");
                super.updateDisplayList(unscaledWidth, unscaledHeight);
            }

            private var cornerRadius:Number = 2;
        ]]>
    </fx:Script>

    <!-- states -->
    <s:states>
        <s:State name="up" />
        <s:State name="over" />
        <s:State name="down" />
        <s:State name="disabled" />
    </s:states>

    <s:Rect id="blueRect" radiusX="8" radiusY="8" top="0" right="0" bottom="0" left="0"
minHeight="30">
        <s:fill>
            <s:LinearGradient x="0" y="0" scaleX="44" rotation="90">
                <s:GradientEntry color="#3399ff" ratio="0" color.over="#66CCFF"/>
                <s:GradientEntry color="#3366cc" ratio="1" color.over="#3399CC"/>
            </s:LinearGradient>
        </s:fill>
        <s:stroke>
            <s:SolidColorStroke color="#ffffff" weight="2"/>
        </s:stroke>
    </s:Rect>
    <!-- Border -->
    <s:Rect radiusX="6" radiusY="6"
        top="2" right="2" height="15" left="2">
        <s:fill>
            <s:LinearGradient x="0" y="0" scaleX="23" rotation="90">
                <s:GradientEntry color="#ffffff" ratio="0" alpha=".3"/>
                <s:GradientEntry color="#ffffff" ratio="1" alpha=".1"/>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <s:Image id="myImage"
        height="0" height.over="90"
        source="@Embed(source='../../assets/butterfly.png')"
        left="20"/>
    <s:Label id="labelDisplay"
        visible.over="false"
        textAlign="center" verticalAlign="middle"
        maxDisplayedLines="1"
        horizontalCenter="0" verticalCenter="1"
        left="10" right="10" top="2" bottom="2"/>

</s:SparkSkin>
```

More Help topics

"Transitions" on page 1870

"Spark effects" on page 1805

Subcomponent skinning

Some components are composites of other components. They use other components as part of their user interface. For example, a NumericStepper consists of a Button control with a down arrow, a Button control with an up arrow, and a TextInput control that displays the current value. The components that make up a composite component's user interface are known as subcomponents.

To skin Spark subcomponents you edit the skin parts in the skin class. Each subcomponent corresponds to a skin part that is defined on the host component. For example, to customize the button skin parts in a NumericStepperSkin class, you can set the value of their `skinClass` property to a custom class.

In many cases, the subcomponents that are defined in the main skin have their own skin classes. This is common if the subcomponents need to define their own appearance based on their state. For example, the buttons in a NumericStepper should react to their own `up` and `down` states rather than the `up` and `down` states of the composite component.

When skinning subcomponents, you typically use the SparkSkin class rather than the Skin class as the root. This is because Spark components usually rely on the colorize exclusions and other properties that require the SparkSkin class.

The following example defines custom subcomponent skins for the up and down buttons in a NumericStepper composite skin:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/NumericStepperSkinPartExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="200" width="200">

    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <s:SkinnableContainer id="myContainer"
        height="200" width="200">
        <s:NumericStepper skinClass="mySkins.NumericStepperSkin"/>
    </s:SkinnableContainer>
</s:Application>
```

The following composite skin class is nearly identical to the default NumericStepperSkin except that it defines custom skins for the button subcomponents. In this case, the spark.skins.spark.SpinnerDecrementButtonSkin and spark.skins.spark.SpinnerIncrementButtonSkin classes are replaced with mySkins.SpinnerDownButton and mySkins.SpinnerUpButton respectively.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/NumericStepperSkin.mxml -->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minHeight="24"
    alpha.disabled="0.5">
    <fx:Metadata>
        [HostComponent("spark.components.NumericStepper")]
    </fx:Metadata>

    <fx:Script>
        /*
           Define the skin elements that should not be colorized.
          For numeric stepper, the skin itself is colorized but the individual parts are not.
        */
        static private const exclusions:Array = ["textInput", "decrementButton",
"incrementButton"];
        override public function get colorizeExclusions():Array {return exclusions;}
    </fx:Script>

    <s:states>
        <s:State name="normal"/>
        <s:State name="disabled"/>
    </s:states>
    <s:Button id="incrementButton" right="0" top="0" height="50%"
            skinClass="mySkins.SpinnerUpButton"/>
    <s:Button id="decrementButton" right="0" bottom="0" height="50%"
            skinClass="mySkins.SpinnerDownButton"/>

    <s:TextInput id="textDisplay" left="0" top="0" right="18" bottom="0"
        skinClass="spark.skins.spark.NumericStepperTextInputSkin"/>

</s:SparkSkin>
```

The following skin classes replace the SpinnerIncrementButtonSkin and SpinnerDecrementButtonSkin classes in the custom NumericStepperSkin example. Instead of drawing arrows, these custom skins use + and - signs for incrementing and decrementing the value of the TextInput control.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/SpinnerDownButton.mxml -->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <s:states>
        <s:State name="up" />
        <s:State name="over"/>
        <s:State name="down" />
        <s:State name="disabled" />
    </s:states>

    <!-- border/fill -->
    <s:Path data="M 0 0 h 18 v 8 Q 18 9 16 10 h -16 Z"
          left="0" top="0" right="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0x686868" weight="1"/>
        </s:stroke>
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0xE8E8E8"
                               color.over="0xC2C2C2"
                               color.down="0xAEB0B1" />
                <s:GradientEntry color="0xDFDFDF"
                               color.over="0xADAEAF"
                               color.down="0xA1A3A5" />
            </s:LinearGradient>
        </s:fill>
    </s:Path>

    <!-- highlight -->
    <s:Path data="M 0 0 h 16 v 6 Q 16 8 14 8 h -14 Z"
          left="1" top="1" right="1" bottom="1" >
        <s:stroke>
            <s:LinearGradientStroke rotation="90" weight="1">
                <s:GradientEntry color="0xFFFFFF"
                               color.down="0x000000"
                               alpha="0.55"
                               alpha.over="0.55"
                               alpha.down="0.15" />
                <s:GradientEntry color="0xFFFFFF"
                               color.down="0x000000"
                               alpha="0.2475"
```

```
                              alpha.over="0.2475"
                              alpha.down="0" />
            </s:LinearGradientStroke>
        </s:stroke>
    </s:Path>

    <!-- shadow -->
    <s:Rect left="1" top="2" right="1" height="1" includeIn="down">
        <s:fill>
            <s:SolidColor color="0x000000" alpha="0.07" />
        </s:fill>
    </s:Rect>

    <!-- Replace the down arrow with a minus sign. -->
    <s:Label id="downArrow"
        text="-"
        horizontalCenter="0"
        verticalCenter="0">
    </s:Label>
</s:SparkSkin>
```

```
<?xml version="1.0" encoding="utf-8"?>
<!-- SparkSkinning/mySkins/SpinnerUpButton.mxml -->
<s:SparkSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Metadata>
        [HostComponent("spark.components.Button")]
    </fx:Metadata>

    <fx:Script>
    </fx:Script>

    <s:states>
        <s:State name="up" />
        <s:State name="over"/>
        <s:State name="down" />
        <s:State name="disabled" />
    </s:states>

    <!-- border/fill -->
    <s:Path data="M 0 0 h 18 v 8 Q 18 9 16 10 h -16 Z"
          left="0" top="0" right="0" bottom="0">
        <s:stroke>
            <s:SolidColorStroke color="0x686868" weight="1"/>
        </s:stroke>
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0xE8E8E8"
                              color.over="0xC2C2C2"
                              color.down="0xAEB0B1" />
                <s:GradientEntry color="0xDFDFDF"
                              color.over="0xADAEAF"
                              color.down="0xA1A3A5" />
            </s:LinearGradient>
        </s:fill>
    </s:Path>

    <!-- highlight -->
    <s:Path data="M 0 0 h 16 v 6 Q 16 8 14 8 h -14 Z"
          left="1" top="1" right="1" bottom="1" >
        <s:stroke>
            <s:LinearGradientStroke rotation="90" weight="1">
                <s:GradientEntry color="0xFFFFFF"
                               color.down="0x000000"
                               alpha="0.55"
                               alpha.over="0.55"
                               alpha.down="0.15" />
                <s:GradientEntry color="0xFFFFFF"
                               color.down="0x000000"
                               alpha="0.2475"
                               alpha.over="0.2475"
                               alpha.down="0" />
            </s:LinearGradientStroke>
        </s:stroke>
    </s:Path>

    <!-- shadow -->
    <s:Rect left="1" top="2" right="1" height="1" includeIn="down">
        <s:fill>
            <s:SolidColor color="0x000000" alpha="0.07" />
        </s:fill>
    </s:Rect>

    <!-- Replace the up arrow with a plus sign. -->
    <s:Label id="upArrow"
        text="+"
        horizontalCenter="0"
        verticalCenter="0">
    </s:Label>
</s:SparkSkin>
```

Packaging skins

You can package custom Spark skins as a SWC file and distribute that SWC file to anyone interested in using your library of skins. The SWC file is also known as a theme SWC file. Theme SWC files for Spark skins typically consist of the following files:

- defaults.css
- One or more skin classes

The defaults.css file file can apply the Spark skin classes and any other style properties. The following sample CSS file from a theme SWC file applies a style property and custom skins to the Button and CheckBox controls in the Spark namespace:

```
@namespace s "library://ns.adobe.com/flex/spark";
s|Button {
    color: Green;
    skinClass: ClassReference("ButtonTransitionSkin");
}
s|CheckBox {
    color: Green;
    skinClass: ClassReference("CheckBoxTransitionSkin");
}
```

To include classes that are compiled, such as Spark skins, in a theme SWC file, you use the `include-classes` compiler option. To include files that are not compiled, such as a stylesheet, you use the include-file compiler option. All files included in a theme SWC file must be in the source path. You use the `output` compiler option to specify the location of the resulting SWC file.

The following command line compiles a new theme SWC file. This theme SWC file includes two Spark skins and a defaults.css file:

```
compc -source-path c:/temp/myskins
    -include-classes ButtonTransitionSkin CheckBoxTransitionSkin
    -include-file defaults.css c:/temp/myskins/defaults.css
    -output c:/temp/myskins/MySkins.swc
```

To use the theme SWC file in your application, you use the `theme` compiler option, as the following example shows:

```
mxmlc -theme=c:/temp/myskins/MySkins.swc c:myapps/ThemeExample.mxml
```

More Help topics

"Creating a theme SWC file" on page 1565

"Using compc, the component compiler" on page 2194

Skinning MX components

You add skins to Adobe® Flex® components by using ActionScript classes, MXML components, and image files. Image files can contain JPEG, GIF, and PNG images or SWF files.

This topic describes the process for skinning MX components. These are typically components that use the Halo theme. For information about skinning Spark components, see "Spark Skinning" on page 1602.

By default, MX components in a Flex 4 application use the Spark skin classes in the mx.skins.spark.* package. To use Halo skins, you can either apply the Halo theme to your application or set the `compatibility-version` compiler option to 3.0.0.

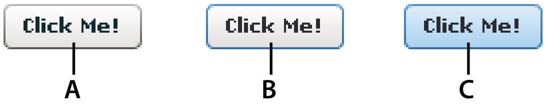
About MX component skinning

Skinning is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of bitmap images, SWF files, or class files that contain drawing methods that define vector images.

Skins can define the entire appearance, or only a part of the appearance, of a component in various states. For example, an MX Button control that uses the Halo theme has eight possible states, and eight associated skin properties, as the following example shows:

State	**Skin property**	**Default skin class**
down	`downSkin`	mx.skins.halo.ButtonSkin
over	`overSkin`	mx.skins.halo.ButtonSkin
up	`upSkin`	mx.skins.halo.ButtonSkin
disabled	`disabledSkin`	mx.skins.halo.ButtonSkin
selectedDisabled	`selectedDisabledSkin`	mx.skins.halo.ButtonSkin
selectedDown	`selectedDownSkin`	mx.skins.halo.ButtonSkin
selectedOver	`selectedOverSkin`	mx.skins.halo.ButtonSkin
selectedUp	`selectedUpSkin`	mx.skins.halo.ButtonSkin

The default skins for the up, over, and down states appear as follows:



***A.** up **B.** over **C.** down*

Other controls have similar states with associated skins. For example, RadioButton controls, which are subclasses of Button, also have up, down, and over skins. The ComboBox control has skins the define the appearance of the control when it is in the disabled, down, and over states.

You create a skin by using a bitmap image, a SWF file, or a class defined in ActionScript or in MXML. All components have a default skin class that can represent more than one state of the component. As you can see in the previous table, the eight states of the Button control use the same default skin class, mx.skins.halo.ButtonSkin, to draw the skin. Logic within the class determines the appearance of the Button control based on its current state.

You assign a skin to a component by using style properties. You can set a style property by using MXML tag properties, the StyleManager class, `<fx:Style>` blocks, or style sheets. Most application use style sheets to organize and apply skins. Style sheets can be loaded at compile time or at run time. For information on loading style sheets at run time, see "Loading style sheets at run time" on page 1547.

Types of MX component skins

You typically define a skin for the Halo theme as a bitmap graphic or as a vector graphic. Bitmap graphics, called *graphical skins* in Flex, are made up of individual pixels that together form an image. The downside of a bitmap graphic is that it is typically defined for a specific resolution and, if you scale or transform the image, you might notice a degradation in image quality.

A vector graphic, called a *programmatic skin* in Flex, consists of a set of line definitions that specify a line's starting and end point, thickness, color, and other information required by Adobe® Flash® Player to draw the line. When a vector graphic is scaled, rotated, or modified in some other way, it is relatively simple for Flash Player to calculate the new layout of the vector graphic by transforming the line definitions. Therefore, you can perform many types of modifications to vector graphics without noticing any degradation in quality.

One advantage of programmatic skins is you can create vector graphics that allow you a great deal of programmatic control over the skin. For example, you can control the radius of a Button control's corners by using programmatic skins, something you cannot do with graphical skins. You can develop programmatic skins directly in your Flex authoring environment or any text editor, without using a graphics tool such as Adobe Flash. Programmatic skins also tend to use less memory because they contain no external image files.

The following table describes the different types of skins:

Skin type	**Description**
Graphical skins	Images that define the appearance of the skin. These images can JPEG, GIF, or PNG files, or they can be symbols embedded in SWF files. Typically you use drawing software such as Adobe® PhotoShop® or Adobe® Illustrator® to create graphical skins. For more information, see "Creating graphical skins for MX components" on page 1671.
Programmatic skins	ActionScript or MXML classes that define a skin. To change the appearance of controls that use programmatic skins, you edit an ActionScript or MXML file. You can use a single class to define multiple skins. For more information, see "Creating programmatic skins for MX components" on page 1674.
Stateful skins	A type of programmatic skin that uses view states, where each view state corresponds to a state of the component. The definition of the view state controls the look of the skin. Since you can have multiple view states in a component, you can use a single component to define multiple skins. For more information, see "Creating stateful skins for MX components" on page 1687.

Creating graphical skins for MX components

When using graphical skins in the Halo theme, you must embed the image file for the skin in your application. To specify your skin, you can use the `setStyle()` method, set it inline, or use Cascading Style Sheets (CSS), as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/SimpleButtonGraphicSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            upSkin: Embed("../assets/orb_up_skin.gif");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

For information on setting skins by using the `setStyle()` method or by setting them inline, see "Applying MX component skins" on page 1663.

You can assign the same graphic or programmatic skin to two or more skins so that the skins display the same image, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/SimpleButtonGraphicSkinTwoSkins.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            upSkin: Embed("../assets/orb_up_skin.gif");
            overSkin: Embed("../assets/orb_up_skin.gif");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

For more information, see “Creating graphical skins for MX components” on page 1671.

Creating programmatic skins for MX components

You can define programmatic skins for the Halo theme in either MXML or ActionScript. When using programmatic skins, you can define a class for each state of the component, or define a single class for multiple skins. In the following example you create a custom ActionScript skin to define the skin for the up state of the Button control:

```
package {
  // skins\ButtonUpSkinAS.as
  import mx.skins.ProgrammaticSkin;
  public class ButtonUpSkinAS extends ProgrammaticSkin {
    // Constructor.
    public function ButtonUpSkinAS() {
      super();
    }
    override protected function updateDisplayList(unscaledWidth:Number,
      unscaledHeight:Number):void
    {
      graphics.lineStyle(1, 0x0066FF);
      graphics.beginFill(0x00FF00, 0.50);
      graphics.drawRoundRect(0, 0, unscaledWidth, unscaledHeight, 10, 10);
    }
  }
}
```

In ActionScript, you typically use the ProgrammaticSkin class as the base class for your skin. In the skin class, you must override the `updateDisplayList()` method to draw the skin. You then assign the skin component to the appropriate skin property of the Button control using a style sheet, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplySimpleButtonSkinAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            upSkin: ClassReference("ButtonUpSkinAS");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

For information on applying skins, see “Applying MX component skins” on page 1663.

In the following example you create a custom MXML component to define the skin for the up state of the Button control. In MXML, you use ProgrammaticSkin as the base class of your skin. This is the same skin defined above in ActionScript:

```
<?xml version="1.0"?>
<!-- skins\ButtonUpMXMLSkin.mxml -->
<skins:ProgrammaticSkin
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:skins="mx.skins.*">
  <fx:Script>
    <![CDATA[
      override protected function updateDisplayList(unscaledWidth:Number,
        unscaledHeight:Number):void
      {
        graphics.lineStyle(1, 0x0066FF);
        graphics.beginFill(0x00FF00, 0.50);
        graphics.drawRoundRect(0, 0, unscaledWidth, unscaledHeight, 10, 10);
      }
    ]]>
  </fx:Script>
</skins:ProgrammaticSkin>
```

Notice that you include the `xmlns:skins="mx.skins.*"` namespace declaration in MXML. This is necessary because by default you cannot use the ProgrammaticSkin class as the base class of an MXML component.

In the following example, you create a component that defines skins for multiple Button states. In this example, you use a `case` statement to determine the current state of the Button control based on the `name` property of the skin, where the `name` property contains the current name of the skin. For example, if you define a programmatic skin for a Button control, the `name` property could be any of the skin states: `downSkin`, `upSkin`, `overSkin`, `disabledSkin`, `selectedDisabledSkin`, `selectedDownSkin`, `selectedOverSkin`, or `selectedUpSkin`.

```
package {
  // skins\ButtonUpAndOverSkinAS.as
  import mx.skins.ProgrammaticSkin;
  public class ButtonUpAndOverSkinAS extends ProgrammaticSkin {
    // Constructor.
    public function ButtonUpAndOverSkinAS() {
      super();
    }
    override protected function updateDisplayList(unscaledWidth:Number,
      unscaledHeight:Number):void
    {
      switch (name)
      {
       case "upSkin": {
        graphics.lineStyle(1, 0x0066FF);
        graphics.beginFill(0x00FF00, 0.50);
        graphics.drawRoundRect(0, 0, unscaledWidth, unscaledHeight, 10, 10);
       }
       case "overSkin": {
        graphics.lineStyle(1, 0x0066FF);
        graphics.beginFill(0x00CCFF, 0.50);
        graphics.drawRoundRect(0, 0, unscaledWidth, unscaledHeight, 10, 10);
       }
      }
    }
  }
}
```

You then assign the ActionScript class to the appropriate skin properties:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonUpOverSkinAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            upSkin: ClassReference("ButtonUpAndOverSkinAS");
            overSkin: ClassReference("ButtonUpAndOverSkinAS");
        }
  </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

For more information on creating programmatic skins, see "Creating programmatic skins for MX components" on page 1674.

Creating stateful skins for MX components

When using a stateful skin with the Halo theme, you define a skin that defines a view state for each state of a component.

You typically define a stateful skin as a subclass of UIComponent, in ActionScript or in MXML, because the view state mechanism is built into the UIComponent class. For more information, see "Creating stateful skins for MX components" on page 1687.

When using stateful skins, you must explicitly specify the default skin for all states not defined by the stateful skin component.

Sizing MX component skins

Before a component applies a skin, it first determines its size without the skin. The component then examines the skin to determine whether the skin defines a specific size. If not, the component scales the skin to fit. If the skin defines a size, the component sizes itself to the skin.

Most skins do not define any size constraints, which allows the component to scale the skin as necessary. For more information on writing skins that contain size information, see "Implementing the measuredWidth and measuredHeight getters" on page 1684.

Skinning MX subcomponents

In some cases, you want to reskin subcomponents. The following example reskins the vertical MX ScrollBar control that appears in List controls:

```
<?xml version="1.0"?>
<!-- skins/SubComponentSkins.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script><![CDATA[
     [Bindable]
     private var theText:String = "Lorem ipsum dolor sit amet, consectetur " +
        "adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore " +
        "magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco " +
        "laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor " +
        "in reprehenderit in voluptate velit esse cillum dolore eu fugiat " +
        "nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt " +
        "in culpa qui officia deserunt mollit anim id est laborum.";
  ]]></fx:Script>
  <fx:Style>
     .myScrollStyle {
        upArrowUpSkin: Embed("../assets/uparrow_up_skin.gif");
        downArrowUpSkin: Embed("../assets/downarrow_up_skin.gif");
     }
  </fx:Style>
  <mx:TextArea id="ta1"
     width="400"
     height="50"
     verticalScrollPolicy="on"
     verticalScrollBarStyleName="myScrollStyle"
     text="{theText}"
  />
</s:Application>
```

By setting the value of the `verticalScrollBarStyleName` property in the List type selector, all vertical ScrollBar controls in List components have a custom skin. The ScrollBar controls in other parts of the application do not have the custom skin.

As with all style sheets, you can define the skins in a separate CSS file and use the `source` property of the `<fx:Style>` tag to point to that file; for example:

```
<fx:Style source="../stylesheets/MySkins.css"/>
```

Creating a Halo theme

A *theme* is a collection of style definitions and skins that define the look and feel of an application built with Flex. Theme files can include both graphical and programmatic skins, as well as style sheets.

A theme takes the form of a SWC file that can be applied to an application. You compile a SWC file using the compc command-line compiler utility.

By compiling a SWC file and then using that SWC file as a theme in your application, you remove the burden of compiling all the skin files when you compile your main application. This can make compilation faster and make problems with the application code easier to debug. In addition, a SWC file is easier to transplant onto another application than are a set of classes and image files.

Halo, the default theme set that shipped with Flex 3, is almost entirely made up of programmatic skins, although there are some static graphic elements. Flex includes programmatic and graphical skins that use the Halo look and feel. You can edit the skins by using either the programmatic or graphical technique to reskin MX components for the Halo theme.

To use the Halo theme in a Flex 4 application, you can use the `theme` compiler option, or you can set the `compatibility-version` option to 3.0.0. If you use the Halo theme, then the Halo theme is applied to all MX components in your application. The Spark components continue to use the Spark theme unless you specifically override them.

For more information on creating and applying themes, see "About themes" on page 1561.

MX component skin resources

Flex includes the following graphical and programmatic source files for MX component skins:

Base skin classes in the mx.skins.* package These abstract skin classes define the basic functionality of Halo skin classes for MX components in Flex. For more information, see "Creating programmatic skins for MX components" on page 1674.

Programmatic skins in the mx.skins.halo.* package These concrete skin classes extend the base skin classes in the mx.skins.* package. They define the appearance of skins for MX components in the Halo theme. You can extend or edit these skins to create new programmatic skins based on the default Flex look and feel. For more information, see "Creating programmatic skins for MX components" on page 1674.

Spark skins in the mx.skins.spark.* package These skin classes define the default appearance of MX components in a Flex 4 application. For more information, see "Spark Skinning" on page 1602.

Graphical Aeon theme The Aeon theme for MX components includes the AeonGraphical.css file and the AeonGraphical.swf file that defines the skin symbols. These are in the framework/themes directory. In addition, Flex includes the FLA source file for the AeonGraphical.swf file. For more information, see "About themes" on page 1561.

You use these files to create skins for MX components based on the Flex look and feel, or create your own.

Creating MX component skins using Adobe tools

You can create graphical, programmatic, and stateful skins for the Halo theme by using Adobe® Flash® CS3, Adobe Illustrator CS3, Adobe Photoshop CS3, and Adobe® Fireworks® CS3. To create skins by using these tools, first download and install extensions that simplify the skinning process. These extensions include the skin templates for all components. For more information on using these tools to create skins, and to download the required extensions, see http://www.adobe.com/go/flex3_cs3_skinning_extensions.

Creating skins in Adobe Flash CS3

While the Flex skinning extensions provide tools that make it easy to create skins in Flash CS3, such as the template of all Flex skins, you are not required to use them to create skins in Flash CS3. Flex lets you use the `[Embed]` metadata tag to import a skin class into an application from a SWC file created in Flash CS3.

Creating a skin class in Flash CS3

1 In Flash CS3, select File > New to open the New Document dialog box.

2 Select Flash File (ActionScript 3.0) as the file type.

3 Create a movie clip symbol for the skin. The symbol name, for example Button_upSkin, becomes the class name of the skin.

Set the following properties on the movie clip symbol:

- Set the Type to Movie clip.
- Set the Base class to flash.display.MovieClip.
- Set the Export for ActionScript option.

- Set the Export in first frame option.
- Set the Class name to be the same as the symbol name.
- Optionally set Enable guides for 9-slice scaling.

4 Save the .fla file.

5 Use File > Publish Settings to open the Publish Settings dialog box.

6 Select Export SWC in the Flash tab in the Publish Settings dialog box.

7 Select File > Publish to publish your skin as a SWC file in the same directory as the .fla file.

Embed the skin class in an application

1 Add the SWC file to the library path of your application:

- In Flash Builder, use the Projects > Properties command to specify the location of the SWC file, or copy it to the libs directory of your Flash Builder project.
- For the mxmlc command-line compiler, use the `library-path` option to specify the location of the SWC file.

2 Use the `[Embed]` metadata tag to embed the skin in your application. You can use the `[Embed]` tag in an `<fx:Style>` block of an MXML file, or in a CSS file.

The `[Embed]` metadata tag has the following syntax for skin classes created in Flash CS3:

```
[Embed(skinClass="className")]
```

For example, if you create a skin named Button_upSkin for the up skin of the Flex Button control, embed the skin as the following example shows:

```
<fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    s|Button
    {
        upSkin: [Embed(skinClass="Button_upSkin")]
    }
</fx:Style>
```

3 Compile and run your application.

Applying MX component skins

You apply skins by using CSS, by specifying them inline in MXML, by calling the `setStyle()` method, or by using the StyleManager class.

Applying graphical skins inline

When you apply a skin inline, you specify it as the value of a skin style in MXML.

To apply a graphical skin inline, you embed the skin by using the appropriate skin property of the control, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesInline.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Button id="b1"
        label="Click Me"
        overSkin="@Embed(source='../assets/orb_over_skin.gif')"
        upSkin="@Embed(source='../assets/orb_up_skin.gif')"
        downSkin="@Embed(source='../assets/orb_down_skin.gif')"/>

</s:Application>
```

The location of the skin asset is relative to the location of the MXML file that embeds it.

When embedding inline, you use `@Embed` (with an at [@] sign prefix) rather than `Embed`, which you use in CSS files.

Applying programmatic skins inline

For programmatic skins, you specify the class name for each state that you reskin and enclose the class name with curly braces { }. The skin's class definition must be in your source path when you compile the application. If the class file is in the same directory as the application, then you do not need to add it to your source path.

The following example applies the SampleButtonSkin.mxml programmatic skin to the Button control's `upSkin`, `overSkin`, `downSkin`, and `disabledSkin` states:

```
<?xml version="1.0"?>
<!-- skins/ApplyProgrammaticSkinsInline.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Button id="b1"
        label="Click Me"
        overSkin="{ButtonStatesSkin}"
        upSkin="{ButtonStatesSkin}"
        downSkin="{ButtonStatesSkin}"
        disabledSkin="{ButtonStatesSkin}"/>
</s:Application>
```

In this example, the SampleButtonSkin.mxml file is in the same directory as the application. If the skin class is in another directory, you must import the class into the application so that the compiler can resolve the class name, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyProgrammaticSkinsInlinePackage.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        import myComponents.*;
    </fx:Script>
    <mx:Button id="b1"
        label="Click Me"
        overSkin="{myComponents.ButtonStatesSkin}"
        upSkin="{myComponents.ButtonStatesSkin}"
        downSkin="{myComponents.ButtonStatesSkin}"
        disabledSkin="{myComponents.ButtonStatesSkin}"/>
</s:Application>
```

When you define stateful skin, you set the `skin` style property of the control to the class name of your skin component, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatefulSkinInlineAll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        import myComponents.*;
    </fx:Script>
    <mx:Button id="b"
        label="Hello"
        skin="{myComponents.MyButtonStatefulSkinAll}"/>

</s:Application>
```

Applying MX component skins by using CSS

When applying skins with CSS, you can use type or class selectors so that you can apply skins to one component or to all components of the same type. You can define the style sheet in the body of the `<fx:Style>` tag or reference an external style sheet, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/UseHaloSkinsStyleSheet.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Style source="../assets/UseHaloSkins.css"/>
    <mx:CheckBox id="cb1" label="Click Me"/>

    <mx:RadioButton id="rb1" label="Click Me"/>
</s:Application>
```

You can apply skins to a single instance of a component by defining a class selector. The following example applies the custom style to the second button only:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesClassSelector.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Style>
        .myButtonStyle {
            overSkin: Embed("../assets/orb_over_skin.gif");
            upSkin: Embed("../assets/orb_up_skin.gif");
            downSkin: Embed("../assets/orb_down_skin.gif");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
    <mx:Button id="b2" label="Click Me" styleName="myButtonStyle"/>
</s:Application>
```

You can load style sheets at run time by compiling them into a SWF file. You then use the top-level StyleManager class's `loadStyleDeclarations()` method to load the CSS-based SWF file at run time, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/BasicApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script>
        <![CDATA[
        public function applyRuntimeStyleSheet():void {
            styleManager.loadStyleDeclarations("assets/BasicStyles.swf")
        }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label text="Click the button to load a new CSS-based SWF file."/>
        <s:Button id="b1" label="Click Me" click="applyRuntimeStyleSheet()"/>
    </s:VGroup>

</s:Application>
```

For more information, see “Loading style sheets at run time” on page 1547.

Using CSS to apply graphical skins to MX components

You typically embed graphical skins as properties of a CSS file in the `<fx:Style>` tag or in an external style sheet, just as you would apply any style property, such as `color` or `fontSize`. The following example defines skins on the Button type selector. In this example, all Buttons controls get the new skin definitions.

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesTypeSelector.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";

        mx|Button {
            overSkin: Embed("../assets/orb_over_skin.gif");
            upSkin: Embed("../assets/orb_up_skin.gif");
            downSkin: Embed("../assets/orb_down_skin.gif");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

When you embed the graphical skin, you embed it into your application's SWF file. For more information on embedding assets, see "Embedding assets" on page 1699.

Using CSS to apply programmatic skins to MX components

You apply programmatic skins in a CSS file by using the `ClassReference` directive. This directive takes a class name as the argument, where the class name corresponds to the ActionScript or MXML file containing your skin definition. The skin's class must be in your source path when you compile the application.

When using programmatic skins, you assign the component name to the associated skin property. In the following example, the skin is in the file myComponents/MyButtonSkin.as where myComponents is a subdirectory of the directory containing your application:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            upSkin: ClassReference("myComponents.MyButtonSkin");
        }
    </fx:Style>
    <mx:Button label="Hello World" id="b" />
</s:Application>
```

When you define stateful skin, you set the `skin` style property of the control to the class name of your skin component, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatefulSkinAll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            skin: ClassReference("myComponents.MyButtonStatefulSkinAll");
        }
    </fx:Style>
    <mx:Button label="Hello" id="b" />
</s:Application>
```

Applying MX component skins with the setStyle() method

Skins are defined as style properties, therefore, you can access them with the setStyle() and `getStyle()` methods. This lets you change skins during run time, or dynamically define them, as long as you embed the graphical asset at compile time.

For more information on using the `setStyle()` method, see "Using the setStyle() and getStyle() methods" on page 1540.

Using the setStyle() method to apply graphical skins to MX components

To embed an image so that you can use it with the `setStyle()` method, you use the `[Embed]` metadata tag and assign a reference to a variable. You can then use the `setStyle()` method to apply that image as a skin to a component.

The following example embeds three images and applies those images as skins to an instance of a Button control:

```
<?xml version="1.0"?>
<!-- skins/EmbedWithSetStyle.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="init();">
    <fx:Script>
        <![CDATA[
            [Embed("../assets/orb_over_skin.gif")]
            public var os:Class;

            [Embed("../assets/orb_down_skin.gif")]
            public var ds:Class;

            [Embed("../assets/orb_up_skin.gif")]
            public var us:Class;

            private function init():void {
                b1.setStyle("upSkin", us);
                b1.setStyle("overSkin", os);
                b1.setStyle("downSkin", ds);
            }
        ]]>
    </fx:Script>

    <mx:Button label="Click Me" id="b1"/>
</s:Application>
```

Using the setStyle() method to apply programmatic skins to MX components

For programmatic skins, you apply a skin to a control by using the setStyle() method. The skin component must be in your source path when you compile the application. The following example applies the ButtonStatesSkin.as component to an MX Button by using the `setStyle()` method:

```
<?xml version="1.0"?>
<!-- skins/ApplyProgrammaticSkinsSetStyle.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            public function changeSkins():void {
                if (cb1.selected) {
                    b1.setStyle("upSkin", ButtonStatesSkin);
                    b1.setStyle("downSkin", ButtonStatesSkin);
                    b1.setStyle("overSkin", ButtonStatesSkin);
                    b1.setStyle("disabledSkin", ButtonStatesSkin);
                } else {
                    b1.setStyle("upSkin", null);
                    b1.setStyle("downSkin", null);
                    b1.setStyle("overSkin", null);
                    b1.setStyle("disabledSkin", null);
                }
            }

        ]]>
    </fx:Script>
    <mx:Button id="b1" label="Click Me"/>
    <mx:CheckBox id="cb1" label="Apply custom skin class" click="changeSkins();"/>

</s:Application>
```

The reference to the ButtonStatesSkin class in the `setStyle()` method causes the compiler to link in the entire ButtonStatesSkin class at compile time. The resulting SWF file will be larger than if there were no reference to this class, even if the `changeSkins()` method is never called.

In the previous example, the SampleButtonSkin.mxml file is in the same directory as the application. If the skin class is in another directory, you must import the class into the application so that the compiler can resolve the class name, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyProgrammaticSkinsSetStylePackage.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import myComponents.*;

            public function changeSkins():void {
                if (cb1.selected) {
                    b1.setStyle("upSkin", myComponents.ButtonStatesSkin);
                    b1.setStyle("downSkin", myComponents.ButtonStatesSkin);
                    b1.setStyle("overSkin", myComponents.ButtonStatesSkin);
                    b1.setStyle("disabledSkin", myComponents.ButtonStatesSkin);
                } else {
                    b1.setStyle("upSkin", null);
                    b1.setStyle("downSkin", null);
                    b1.setStyle("overSkin", null);
                    b1.setStyle("disabledSkin", null);
                }
            }
        ]]>
    </fx:Script>
    <mx:Button id="b1" label="Click Me"/>
    <mx:CheckBox id="cb1" label="Apply custom skin class" click="changeSkins();"/>

</s:Application>
```

When you define stateful skin, you use the `setStyle()` method to set the `skin` style property of the control to the class name of your skin component. For more information on applying stateful skins, see "Creating stateful skins for MX components" on page 1687.

Applying MX component skins with the StyleManager class

To apply skins to all instances of a control, you can use the StyleManager class, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/EmbedWithStyleManager.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" initialize="init()">
    <fx:Script>
        <![CDATA[
            [Embed("../assets/orb_over_skin.gif")]
            public var os:Class;
            [Embed("../assets/orb_down_skin.gif")]
            public var ds:Class;
            [Embed("../assets/orb_up_skin.gif")]
            public var us:Class;
            private function init():void {
               styleManager.getStyleDeclaration("mx.controls.Button").setStyle("upSkin", us);
             styleManager.getStyleDeclaration("mx.controls.Button").setStyle("overSkin", os);
             styleManager.getStyleDeclaration("mx.controls.Button").setStyle("downSkin", ds);
            }
        ]]>
    </fx:Script>

    <mx:Button label="Click Me" id="b1"/>

</s:Application>
```

For more information on using the StyleManager class, see “Using the StyleManager class” on page 1537.

Creating graphical skins for MX components

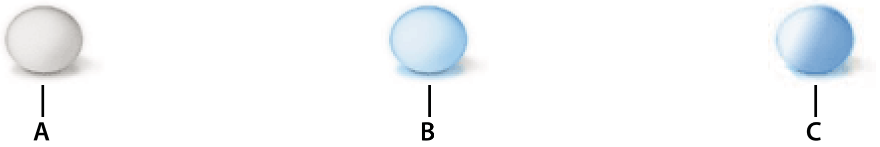
To use graphical skins with MX components, you embed image files in your application. These images can be JPEG, GIF, or PNG files, or they can be symbols embedded in SWF files.

When using SWF files for skins, you can use static assets, which are SWF files that contain symbol definitions but no ActionScript 3.0 code, or use dynamic assets. Dynamic assets correspond to components and contain ActionScript 3.0 code. These components are designed to work with Flex features such as view states and transitions. To use dynamic assets in an application, you export the symbols in the SWF file to a SWC file, and then link the SWC file to your application.

For more information on embedding assets into an application, see “Embedding assets” on page 1699.

Using JPEG, GIF, and PNG files as MX component skins

To use a JPEG, GIF, or PNG file as a skin, you must embed the file in your application. For example, to change the appearance of a Button control, you might create three image files called orb_up_skin.gif, orb_down_skin.gif, and orb_over_skin.gif:



A. orb_down_skin.gif B. orb_over_skin.gif C. orb_up_skin.gif

The following example uses graphical skins for the up, over, and down states of the Button control:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesTypeSelector.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";

        mx|Button {
            overSkin: Embed("../assets/orb_over_skin.gif");
            upSkin: Embed("../assets/orb_up_skin.gif");
            downSkin: Embed("../assets/orb_down_skin.gif");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

The reason that you must embed the file is that in order to determine a component's minimum and preferred sizes, skin assets must be present as soon as the component is created. If you reference external assets at run time, Flex does not have the sizing information and, therefore, cannot render the skins properly.

Because skins are embedded, if you change the graphics files that comprise one or more skins, you must recompile your application for the changes to take effect. For more information on embedding assets, see “Embedding assets” on page 1699.

One drawback to embedding images as skins is that they can become distorted if you resize the component that has a skin. You can use a technique called 9-slice scaling to create skins that do not become distorted when the component is resized. For information on the 9-slice scaling technique, see “Using 9-slice scaling with embedded images” on page 1711.

Using static SWF assets as MX component skins

Static SWF files created in Flash 8 or Flash 9 contain artwork or skins, but do not contain any ActionScript 3.0 code. You can use the entire SWF file as a single skin, or you can use one or more symbols inside the SWF file as a skin. To embed an entire SWF file, you point to the location of the SWF file with the `source` property in the `Embed` statement, as follows:

```
<?xml version="1.0"?>
<!-- skins/EmbedSWFSource.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|Button {
        upSkin: Embed(source="../assets/SubmitButtonUpSkin.swf");
     }
  </fx:Style>

  <mx:Button id="b1"/>
</s:Application>
```

To import a symbol from a SWF file, you use the `symbol` property to specify the symbol name that you want to use in addition to pointing to the location of the SWF file with the `source` property. You must separate each property of the `Embed` statement with a comma. The following example replaces the MX Button control's up, over, and down skins with individual symbols from a SWF file:

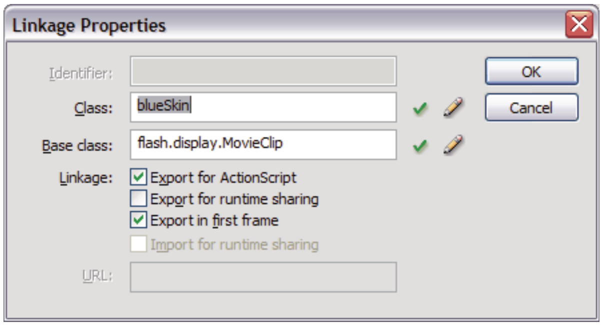
```
<?xml version="1.0"?>
<!-- skins/EmbedSymbolsCSS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
    @namespace mx "library://ns.adobe.com/flex/mx";
    mx|Button {
        upSkin: Embed(source='../assets/SubmitButtonSkins.swf',
            symbol='MyUpSkin');
        overSkin: Embed(source='../assets/SubmitButtonSkins.swf',
            symbol='MyOverSkin');
        downSkin: Embed(source='../assets/SubmitButtonSkins.swf',
            symbol='MyDownSkin');
    }
 </fx:Style>

  <mx:Button id="b1"/>
</s:Application>
```

You use the same syntax when embedding skin symbols inline, as follows:

```
<?xml version="1.0"?>
<!-- skins/EmbedSymbolsInline.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:Button id="b1"
     overSkin="@Embed(source='../assets/SubmitButtonSkins.swf',
        symbol='MyOverSkin')"
     upSkin="@Embed(source='../assets/SubmitButtonSkins.swf',
        symbol='MyUpSkin')"
     downSkin="@Embed(source='../assets/SubmitButtonSkins.swf',
        symbol='MyDownSkin')"/>
</s:Application>
```

In the source FLA file, all symbols that you use must meet the following conditions:

- The symbol must be on the Stage. After you create an image file and convert it to a symbol, you must drag it from the library to the Stage. Flash does not export symbols that are not on the Stage. Alternatively, you can select the Export in First Frame option in the Linkage Properties dialog box.
- The symbol must have been exported for ActionScript with a linkage name. In Flash, you select the Export for ActionScript option in the symbol's Linkage Properties dialog box, as the following example shows:



Linkage Properties

Identifier:

Class: blueSkin

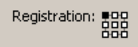
Base class: flash.display.MovieClip

Linkage: ☑ Export for ActionScript
☐ Export for runtime sharing
☑ Export in first frame
☐ Import for runtime sharing

URL:

The linkage name is the name used by Flex. Symbol names are ignored.

- The FLA files cannot contain any ActionScript.
- The symbol must have an upper-left registration point that you select in the Convert to Symbol dialog box. The following example shows an upper-left registration point:


Registration:

You can use 9-slice scaling with image files (a grid with nine regions) so that the skin scales well when the component's size changes. You define the properties for 9-slice scaling of images when you apply the skin in CSS, as the following example shows:

```
@namespace mx "library://ns.adobe.com/flex/mx";
mx|Button {
    overSkin: Embed(
        "../assets/orb_over_skin.gif",
        scaleGridTop=6,
        scaleGridLeft=12,
        scaleGridBottom=44,
        scaleGridRight=49
    );
}
```

For information on embedding assets that use the 9-slice scaling technique, see "Using 9-slice scaling with embedded images" on page 1711.

Using dynamic SWF assets as MX component skins

Dynamic SWF assets are components that you create in Flash CS3 and then import into your application. Since these assets are components, you can use them just as you would a component shipped with Flex or as you would a custom component that you created as an MXML file or as an ActionScript class.

To create a Flash component, you define the symbol in your FLA file and specify the base class of the symbol as mx.flash.UIMovieClip. To integrate components from a dynamic SWF asset created in Flash CS3, you publish your FLA file as a SWC file.

For more information on creating dynamic SWF assets and using them as skins, see http://www.adobe.com/go/flex3_cs3_skinning_extensions and http://www.adobe.com/go/flex3_cs3_swfkit.

Creating programmatic skins for MX components

You create programmatic skins as ActionScript classes or as MXML components for MX components. You use the basic drawing methods of the Flash Graphics (flash.display.Graphics) package, and apply those skins to your Flex controls.

You can modify programmatic skins that come with Flex or create your own. The programmatic skins used by components are in the mx.skins.halo.* package. All of the skins extend one of the following classes: UIComponent, ProgrammaticSkin, Border, or RectangularBorder.

For information on creating your own skins, see "Programmatic skins recipe for MX components" on page 1675.

One type of programmatic skin, called a stateful skin, uses view states. For information on creating stateful skins, see "Creating stateful skins for MX components" on page 1687.

Flex handles much of the overhead of class definition when you use an MXML component, so in some cases you might find it easier to define your skins as MXML components. The only restriction on MXML components is that you cannot define a constructor. Instead, you use an event handler for the `preinitialize` event to perform the work that you do in an ActionScript constructor.

For general information on creating ActionScript classes and MXML components, see *"Custom Flex components" on page 2356*.

Programmatic skins recipe for MX components

At a minimum, a programmatic skin consists of a constructor (for an ActionScript class), an updateDisplayList() method, and a set of getters and setters for the skin's properties. Programmatic skins generally extend one of the classes in the mx.skins package or the UIComponent class.

To see examples of skins that follow the programmatic skin recipe, look at the concrete classes in the mx.skins.halo package. These are the skins that the components use. Those skins follow the same recipe presented here.

The following example is a typical outline of a programmatic skin:

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/MySkinOutline.as

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  // Extend ProgrammaticSkin.
  public class MySkinOutline extends ProgrammaticSkin {
     // Constructor.
     public function MySkinOutline() {
        // Set default values here.
     }
     // Override updateDisplayList().
     override protected function updateDisplayList(w:Number,
        h:Number):void {
        // Add styleable properties here.
        // Add logic to detect component s state and set properties here.
        // Add drawing methods here.
     }
  }
} // Close unnamed package.
```

In your application, you can apply a programmatic skin using the `ClassReference` statement in CSS:

```
<?xml version="1.0"?>
<!-- skins/ApplyMySkinOutline.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|Button {
        overSkin: ClassReference("MySkinOutline");
        upSkin: ClassReference("MySkinOutline");
        downSkin: ClassReference("MySkinOutline");
     }
  </fx:Style>
  <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

Selecting an interface or base class for your skin

Skin classes must implement one or more interfaces. When you create a programmatic skin, you can either create a class that implements the required interfaces, or you can create a subclass of a class that already implements the required interfaces.

Your decision on which interface to implement, or which class to use as the base class of your skin, might depend on the type of skin that you want to create. For example, if you want to create a skin that defines a border, you might create a subclass of Border. If you want to create a stateful skin, you can create subclass of UIComponent or create a class that implement the IStateClient interface.

You can extend the abstract base classes in the mx.skins package or the concrete classes in the mx.skins.halo package. Extending the former gives you greater control over the look and feel of your skins. Extending the latter is a good approach if you use the default behaviors of the components but also want to add extra styling to them.

Some rules to consider:

- A stateful skin must implement either the IStateClient interface or the IProgrammaticSkin interface. The UIComponent class implements the IStateClient interface. The ProgrammaticSkin class implements the IProgrammaticSkin interface.
- A skin passed to any skin property other than a stateful skin property, such as `Button.upSkin` or `NumericStepper.upArrowOverSkin`, must implement the IFlexDisplayObject interface. The UIComponent class and the ProgrammaticSkin class implement the IFlexDisplayObject interface.

Most Halo skins for MX components extend the mx.skins.ProgrammaticSkin class, but you can select any one of the following as a superclass for your skin:

- The ProgrammaticSkin class implements the IFlexDisplayObject, ILayoutManagerClient, IInvalidating, and ISimpleStyleClient interfaces, so it is the easiest and most common superclass to use.
- The Border class extends the ProgrammaticSkin class and adds support for the `borderMetrics` property. Use this class or the RectangularBorder class if your skin defines the component's border.
- The RectangularBorder class extends the Border class and adds support for the `backgroundImage` style.
- The UIComponent class implements the IStateClient interface, making it easy to use for stateful skins. It is also the component that you use when implementing skins in MXML.

Use the following list of steps to create programmatic skins for your Flex controls. Each step is explained in more detail in the following sections.

Create programmatic skins for MX components

1 Select one of the following base classes as a superclass for your programmatic skin:

- Border or a subclass of Border
- ProgrammaticSkin or a subclass of ProgrammaticSkin
- RectangularBorder or a subclass of RectangularBorder
- UIComponent or a subclass of UIComponent

You can also extend one of the concrete classes in the mx.skins.Halo package.

2 Implement the `updateDisplayList()` method. Put all the drawing and styling calls in this method.

For more information, see “Implementing the updateDisplayList() method” on page 1677.

3 For an ActionScript class, implement the constructor.

4 (Optional) Implement getters for the `measuredWidth` and `measuredHeight` properties.

For more information, see “Implementing the measuredWidth and measuredHeight getters” on page 1684.

5 If the skin is a subclass of Border or RectangularBorder, implement a getter for the `borderMetrics` property.

For more information, see “Implementing a getter for the borderMetrics property” on page 1685.

6 (Optional) Make properties styleable.

If you create a skin that has properties that you want users to be able to set with CSS or with calls to the `setStyle()` method, you must add code to your skin class. For more information, see “Making properties styleable” on page 1695.

Compiling programmatic skins for MX components

When you compile an application that uses programmatic skins, you treat programmatic skins as you would treat any ActionScript class or MXML component, which means that you must add the skins to the `source-path` argument of the compiler. If the skins are in the same directory as the MXML file that you are compiling, you set the `source-path` to a period. The following example shows this with the mxmlc command-line compiler:

```
$ ./mxmlc -source-path=. c:/flex/MyApp.mxml
```

If the programmatic skins are not in a package, you must add them to the unnamed package to make them externally visible. Otherwise, mxmlc throws a compiler error. To do this, you surround the class with a package statement, as the following example shows:

```
package { // Open unnamed package.
    import flash.display.*;
    import mx.skins.ProgrammaticSkin;

    public class MySkin extends ProgrammaticSkin {
        ...
    }
} // Close unnamed package.
```

Implementing the updateDisplayList() method

The updateDisplayList() method defines the look of the skin. It is called after the skin’s construction to initially draw the skin, and then is subsequently called whenever the component is resized, restyled, moved, or is interacted with in some way.

You use the Flash Player drawing methods to draw the programmatic skin in the `updateDisplayList()` method. For more information on the drawing methods, see “Drawing programmatically” on page 1680.

When you implement the `updateDisplayList()` method, you must do the following:

- Use the `override` keyword to override the superclass's implementation.
- Set the return type to `void`.
- Declare the method as `protected`.

The `updateDisplayList()` method takes the height and width of the component as arguments. You use the values of these arguments as the boundaries for the region in which you can draw. The method returns `void`.

You use methods of theGraphics class (such as the `lineTo()` and `drawRect()` methods) to render the skin. To ensure that the area is clear before adding the component's shapes, you should call the `clear()` method before drawing. This method erases the results of previous calls to the `updateDisplayList()` method and removes all the images that were created by using previous draw methods. It also resets any line style that was specified with the `lineStyle()` method.

To use the methods of the Graphics package, you must import the flash.display.Graphics class, and any other classes in the flash.display package that you use, such as GradientType or Font. The following example imports all classes in the flash.display package:

```
import flash.display.*;
```

The following example draws a rectangle as a border around the component with the `drawRect()` method:

```
g.drawRect(0, 0, width, height);
```

The following example skin class draws an *X* with a border around it:

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/CheckboxSkin.as

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  public class CheckboxSkin extends ProgrammaticSkin {
     // Constructor.
     public function CheckboxSkin() {
        // Set default values here.
     }
  override protected function updateDisplayList(w:Number, h:Number):void {
     var g:Graphics = graphics;
     g.clear();
     g.beginFill(0xFFFFFF,1.0);
     g.lineStyle(2, 0xFF0000);
     g.drawRect(0, 0, w, h);
     g.endFill();
     g.moveTo(0, 0);
     g.lineTo(w, h);
     g.moveTo(0, h);
     g.lineTo(w, 0);
  }
  }
} // Close unnamed package.
```

For a description of common methods of the Graphics package, see "Drawing programmatically" on page 1680. For details about these methods, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

One common task performed in the `updateDisplayList()` method is to change properties of the skin, depending on the current state of the control. For example, if you define a programmatic skin for a Button control, you can change the border thickness or color of the background when the user moves the mouse over or clicks the Button control.

You check the state by using the `name` property of the skin. The `name` is the current name of the skin. For example, if you define a programmatic skin for a Button control, the `name` property could be any of the skin states: `downSkin`, `upSkin`, `overSkin`, `disabledSkin`, `selectedDisabledSkin`, `selectedDownSkin`, `selectedOverSkin`, or `selectedUpSkin`.

The following example checks which state the Button control is in and adjusts the line thickness and background fill color appropriately. The result is that when the user clicks the Button control, Flex redraws the skin to change the line thickness to 2 points. When the user releases the mouse button, the skin redraws again and the line thickness returns to its default value of 4. The background fill color also changes depending on the Button control's state.

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/ButtonStatesSkin.as
  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  public class ButtonStatesSkin extends ProgrammaticSkin {
     public var backgroundFillColor:Number;
     public var lineThickness:Number;
     // Constructor.
     public function ButtonStatesSkin() {
        // Set default values.
        backgroundFillColor = 0xFFFFFF;
        lineThickness = 4;
     }
     override protected function updateDisplayList(w:Number, h:Number):void {
        // Depending on the skin's current name, set values for this skin.
        switch (name) {
           case "upSkin":
            lineThickness = 4;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "overSkin":
            lineThickness = 4;
            backgroundFillColor = 0xCCCCCC;
            break;
           case "downSkin":
            lineThickness = 2;
            backgroundFillColor = 0xFFFFFF;
```

```
                break;
              case "disabledSkin":
                lineThickness = 2;
                backgroundFillColor = 0xCCCCCC;
                break;
            }
            // Draw the box using the new values.
            var g:Graphics = graphics;
            g.clear();
            g.beginFill(backgroundFillColor,1.0);
            g.lineStyle(lineThickness, 0xFF0000);
            g.drawRect(0, 0, w, h);
            g.endFill();
            g.moveTo(0, 0);
            g.lineTo(w, h);
            g.moveTo(0, h);
            g.lineTo(w, 0);
        }
    }
} // Close unnamed package.
```

If you use a single programmatic skin class to define multiple states of a control, you must apply the skin to all appropriate states of that control in your application; for example:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatesSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            overSkin: ClassReference("ButtonStatesSkin");
            upSkin: ClassReference("ButtonStatesSkin");
            downSkin: ClassReference("ButtonStatesSkin");
        }
    </fx:Style>
    <mx:Button id="b1" label="Click Me"/>
</s:Application>
```

If the skin is a subclass of RectangularBorder, you must also call `super.updateDisplayList()` from within the body of the `updateDisplayList()` method.

Drawing programmatically

You use the drawing methods of the Graphics class to draw the parts of a programmatic skin. These methods let you describe fills or gradient fills, define line sizes and shapes, and draw lines. By combining these very simple drawing methods, you can create complex shapes that make up your component skins.

The following table briefly describes the most commonly used drawing methods in the Graphics package:

Method	Summary
`beginFill()`	Begins drawing a fill; for example: `beginFill(0xCCCCFF,1);` If an open path exists (that is, if the current drawing position does not equal the previous position that you specified in a `moveTo()` method) and it has a fill associated with it, that path is closed with a line, and then filled.
`beginGradientFill()`	Begins drawing a gradient fill. If an open path exists (that is, if the current drawing position does not equal the previous position that you specified in a `moveTo()` method), and it has a fill associated with it, that path is closed with a line, and then filled.
`clear()`	Removes all the drawing output associated with the current object. The `clear()` method takes no arguments.

`curveTo()`	Draws a curve using the current line style; for example:
	`moveTo(500, 500);`
	`curveTo(600, 500, 600, 400);`
	`curveTo(600, 300, 500, 300);`
	`curveTo(400, 300, 400, 400);`
	`curveTo(400, 500, 500, 500);`

`drawCircle()`	Draws a circle after you set the line style and fill. You pass the method the x and y positions of the circle, as well as the radius, as the following example shows: `drawCircle(10,10,50);`
`drawRect()`	Draws a rectangle once you set the line style and fill. You pass the method the x and y positions of the rectangle, as well as the length and width of the rectangle, as the following example shows: `drawRect(10,10,100,20);`
`drawRoundRect()`	Draws a rectangle with rounded corners, after you set the line and fill. You pass the method the x and y position of the rectangle, length and height of the rectangle, and the width and height of the ellipse that is used to draw the rounded corners, as the following example shows: `drawRoundRect(10,10,100,20,9,5)`
`endFill()`	Ends the fill specified by the `beginFill()` or `beginGradientFill()` methods. The `endFill()` method takes no arguments. If the current drawing position does not equal the previous position that you specified in a `moveTo()` method and a fill is defined, the path is closed with a line, and then filled.
`lineStyle()`	Defines the stroke of lines created with subsequent calls to the `lineTo()` and `curveTo()` methods. The following example sets the line style to a 2-point gray line with 100% opacity: `lineStyle(2,0xCCCCCC,1)` You can call the `lineStyle()` method in the middle of drawing a path to specify different styles for different line segments within a path. Calls to the `clear()` method reset line styles back to `undefined`.

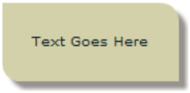
`lineTo()`	Draws a line using the current line style. The following example draws a triangle:
	`moveTo (200, 200);`
	`lineTo (300, 300);`
	`lineTo (100, 300);`
	`lineTo (200, 200);`
	If you call the `lineTo()` method before any calls to the `moveTo()` method, the current drawing position returns to the default value of (0, 0).

`moveTo()`	Moves the current drawing position to the specified coordinates; for example: `moveTo(100,10);`

The following example draws a triangle:

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/CheckBoxAsArrowSkin.as

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  public class CheckBoxAsArrowSkin extends ProgrammaticSkin {

     // Constructor.
     public function CheckBoxAsArrowSkin() {
        // Set default values.
     }
     override protected function updateDisplayList(w:Number, h:Number):void {
        var unscaledHeight:Number = 2;
        var unscaledWidth:Number = 2;
        var arrowColor:Number;
        var g:Graphics = graphics;
        g.clear();
        switch (name) {
           case "upIcon":
           case "selectedUpIcon": {
            arrowColor = 0x666666;
            break;
           }
           case "overIcon":
           case "downIcon":
           case "selectedOverIcon":
           case "selectedDownIcon": {
            arrowColor = 0xCCCCCC;
            break;
           }
        }
        // Draw an arrow.
        graphics.lineStyle(1, 1, 1);
        graphics.beginFill(arrowColor);
        graphics.moveTo(unscaledWidth, unscaledHeight-20);
        graphics.lineTo(unscaledWidth-30, unscaledHeight+20);
        graphics.lineTo(unscaledWidth+30, unscaledHeight+20);
        graphics.lineTo(unscaledWidth, unscaledHeight-20);
        graphics.endFill();
     }
  }
} // Close unnamed package.
```

The ProgrammaticSkin class also defines drawing methods, the most common of which is the drawRoundRect() method. This method programmatically draws a rectangle and lets you set the corner radius, gradients, and other properties. You can use this method to customize borders of containers so that they might appear as the following example shows:



The following code uses the `drawRoundRect()` method to draw this custom VBox border:

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/CustomContainerBorderSkin.as

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.graphics.RectangularDropShadow;
  import mx.skins.RectangularBorder;
  public class CustomContainerBorderSkin extends RectangularBorder {
    private var dropShadow:RectangularDropShadow;
    // Constructor.
    public function CustomContainerBorderSkin() {
    }
    override protected function updateDisplayList(unscaledWidth:Number,
       unscaledHeight:Number):void
    {
       super.updateDisplayList(unscaledWidth, unscaledHeight);
       var cornerRadius:Number = getStyle("cornerRadius");
       var backgroundColor:int = getStyle("backgroundColor");
       var backgroundAlpha:Number = getStyle("backgroundAlpha");
       graphics.clear();
       // Background
       drawRoundRect(0, 0, unscaledWidth, unscaledHeight,
           {tl: 0, tr:cornerRadius, bl: cornerRadius, br: 0},
           backgroundColor, backgroundAlpha);
       // Shadow
       if (!dropShadow)
          dropShadow = new RectangularDropShadow();
       dropShadow.distance = 8;
       dropShadow.angle = 45;
       dropShadow.color = 0;
       dropShadow.alpha = 0.4;
       dropShadow.tlRadius = 0;
       dropShadow.trRadius = cornerRadius;
       dropShadow.blRadius = cornerRadius;
       dropShadow.brRadius = 0;
       dropShadow.drawShadow(graphics, 0, 0, unscaledWidth, unscaledHeight);
    }
  }
}
```

In your application, you apply this skin as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyContainerBorderSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:VBox id="vb1"
     borderSkin="CustomContainerBorderSkin"
     backgroundColor="0xCCCC99"
     backgroundAlpha="0.8"
     cornerRadius="14"
     paddingLeft="20"
     paddingTop="20"
     paddingRight="20"
     paddingBottom="20">
     <mx:Label text="This is a VBox with a custom skin."/>
  </mx:VBox>
</s:Application>
```

The `unscaledWidth` and `unscaledHeight` properties in the previous examples refer to the measurements of the skin as the skin itself understands them. These measurements ignore the fact that external components might have changed the dimensions of the skin. When working inside the component, it is best to use the unscaled measurements.

Implementing the measuredWidth and measuredHeight getters

The `measuredWidth` and `measuredHeight` properties define the default width and height of a component. You can implement getter methods for the `measuredWidth` and `measuredHeight` properties of your skin, but it is not required by most skins. Some skins such as the skins that define the ScrollBar arrows do require that you implement these getters. If you do implement these getters, you must specify the `override` keyword when implementing the superclass's getter methods, and you must make the getters public.

The `measuredWidth` and `measuredHeight` getters typically return a constant number. The application usually honors the measured sizes, but not always. If these getters are omitted, the values of `measuredWidth` and `measuredHeight` are set to the default value of 0.

The following example sets the `measuredWidth` and `measuredHeight` properties to 10, and then overrides the getters:

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/ButtonStatesWithMeasuredSizesSkin.as
  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  public class ButtonStatesWithMeasuredSizesSkin extends ProgrammaticSkin {
     public var backgroundFillColor:Number;
     public var lineThickness:Number;
     private var _measuredWidth:Number;
     private var _measuredHeight:Number;
     // Constructor.
     public function ButtonStatesWithMeasuredSizesSkin() {
        // Set default values.
        backgroundFillColor = 0xFFFFFF;
        lineThickness = 4;
        _measuredHeight = 100;
        _measuredWidth = 150;
     }
     override public function get measuredWidth():Number {
        return _measuredWidth;
     }
     override public function get measuredHeight():Number {
        return _measuredHeight;
     }
     override protected function updateDisplayList(w:Number, h:Number):void {
        // Depending on the skin's current name, set values for this skin.
        switch (name) {
           case "upSkin":
            lineThickness = 4;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "overSkin":
            lineThickness = 4;
            backgroundFillColor = 0xCCCCCC;
            break;
           case "downSkin":
            lineThickness = 2;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "disabledSkin":
            lineThickness = 2;
            backgroundFillColor = 0xCCCCCC;
            break;
        }
        // Draw the box using the new values.
        var g:Graphics = graphics;
        g.clear();
        g.beginFill(backgroundFillColor,1.0);
        g.lineStyle(lineThickness, 0xFF0000);
        g.drawRect(0, 0, w, h);
        g.endFill();
        g.moveTo(0, 0);
        g.lineTo(w, h);
        g.moveTo(0, h);
        g.lineTo(w, 0);
     }
  }
} // Close unnamed package.
```

Implementing a getter for the borderMetrics property

The `borderMetrics` property defines the thickness of the border on all four sides of a programmatic skin. If the programmatic skin is a subclass of Border or RectangularBorder, you must implement a getter for the `borderMetrics` property. Otherwise, this step is optional. This property is of type EdgeMetrics, so your getter must set EdgeMetrics as the return type.

The following example gets the `borderThickness` style and uses that value to define the width of the four sides of the border, as defined in the EdgeMetrics constructor:

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/ButtonStatesWithBorderMetricsSkin.as
  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  import mx.core.EdgeMetrics;
  public class ButtonStatesWithBorderMetricsSkin extends ProgrammaticSkin {
     public var backgroundFillColor:Number;
     public var lineThickness:Number;
     private var _borderMetrics:EdgeMetrics;
     // Constructor.
     public function ButtonStatesWithBorderMetricsSkin() {
        // Set default values.
        backgroundFillColor = 0xFFFFFF;
        lineThickness = 4;
     }
     public function get borderMetrics():EdgeMetrics {
        if (_borderMetrics) {
           return _borderMetrics;
        }
        var borderThickness:Number = getStyle("borderThickness");
        _borderMetrics = new EdgeMetrics(borderThickness,
            borderThickness, borderThickness, borderThickness);
        return _borderMetrics;
     }
     override protected function updateDisplayList(w:Number, h:Number):void
     {
        // Depending on the skin's current name, set values for this skin.
        switch (name) {
           case "upSkin":
            lineThickness = 4;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "overSkin":
            lineThickness = 4;
            backgroundFillColor = 0xCCCCCC;
            break;
           case "downSkin":
            lineThickness = 2;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "disabledSkin":
            lineThickness = 2;
            backgroundFillColor = 0xCCCCCC;
            break;
        }
        // Draw the box using the new values.
        var g:Graphics = graphics;
        g.clear();
        g.beginFill(backgroundFillColor,1.0);
        g.lineStyle(lineThickness, 0xFF0000);
        g.drawRect(0, 0, w, h);
        g.endFill();
        g.moveTo(0, 0);
        g.lineTo(w, h);
        g.moveTo(0, h);
        g.lineTo(w, 0);
     }
  }
} // Close unnamed package.
```

Creating stateful skins for MX components

Many MX components that use the Halo theme, such as Button, Slider, and NumericStepper, support stateful skins. A stateful skin uses view states to specify the skins for the different states of the component. For more information on view states, see “View states” on page 1847.

You can determine whether a skin property supports stateful skins from its description in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For example, all stateful skin properties contain a sentence in the form shown below for the `TitleWindow.closeButtonSkin` property:

“You can use the `closeButtonSkin` style to assign the skin for the following skin states: disabled, down, over, up.”

To function as a stateful skin, the skin must implement the IStateClient interface. Because that interface is implemented by the UIComponent class, you can use any subclass of UIComponent to define a stateful skin. You then assign the stateful skin class to a stateful skin property of the component.

For example, an MX Button control has eight possible states, and eight associated skins. To create a single skin class that defines the skins for all eight states, you create a skin based on the UIComponent control. You then define eight view states within your skin where the name of each view state corresponds to a state of the Button control, as the following example shows:

```
<?xml version="1.0"?>
<mx:UIComponent xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:states>
        <mx:State name="down">
        </mx:State>

        <mx:State name="over">
        </mx:State>

        ...

        <mx:State name="selectedUp">
        </mx:State>
    </mx:states>

    <fx:Script>
        <![CDATA[
            <!-- Define the skin by using the Flash drawing API. -->
        ]]>
    </fx:Script>
</mx:UIComponent>
```

***Note:** You can create a stateful skin in either ActionScript or MXML. The examples in this section use MXML because it requires fewer lines of code to define view states.*

After defining your stateful skin, you assign it to the `skin` style property of the control. You can assign the stateful control by using CSS, the `setStyle()` method, by using inline styles, or by using the StyleManager class. The following example sets it by using CSS:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatefulSkinAll.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
            skin: ClassReference("myComponents.MyButtonStatefulSkinAll");
        }
    </fx:Style>
    <mx:Button label="Hello" id="b" />
</s:Application>
```

You do not have to define all eight skins; you only define the skins that you want to create. For all others, you use the default skins supplied with the theme.

For more information, see “Applying MX component skins” on page 1663.

Example: Creating a stateful Halo skin

A stateful skin is a programmatic skin, so you have to define it using the rules defined in the section “Programmatic skins recipe for MX components” on page 1675. That means you have to define an override of the `updateDisplayList()` method, and for an ActionScript class, you also define a constructor.

To create a view state, you define a base view state, and then define a set of changes, or overrides, that modify the base view state to define each new view state. Each new view state can modify the base state by adding or removing child components, by setting style and property values, or by defining state-specific event handlers.

One of the most common ways to define stateful skins is to define the skin with several properties or styles that can be modified by each view state. For example, the following stateful skin defines a property to control the line weight, fill color, and drop shadow for a skin used by the Button control:

```
<?xml version="1.0"?>
<!-- skins/myComponents/MyButtonStatefulSkin.mxml -->
<mx:UIComponent xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import flash.filters.DropShadowFilter;

      // Define a drop shadow for the over and down states.
      [Bindable]
      private var myFilter:DropShadowFilter = new DropShadowFilter(0);

      // Define a private var for line weight.
      private var _lineWeight:Number = 1;

      // Define public setter and getter for line weight.
      public function get lineWeight():Number
      {
        return _lineWeight;
      }

      public function set lineWeight(value:Number):void
      {
        _lineWeight = value;
        invalidateDisplayList();
      }

      // Define a private var for the fill color.
      private var _rectFill:uint = 0x00FF00;

      // Define public setter and getter for fill color.
      public function get rectFill():uint
      {
        return _rectFill;
      }

      public function set rectFill(value:uint):void
      {
        _rectFill = value;
        invalidateDisplayList();
      }
      override protected function updateDisplayList(unscaledWidth:Number,
          unscaledHeight:Number):void
      {
        graphics.lineStyle(lineWeight, 0x0066FF);
        graphics.beginFill(rectFill, 0.50);
        graphics.drawRoundRect(0, 0, unscaledWidth, unscaledHeight, 10, 10);
        filters = [myFilter];
      }
    ]]>
  </mx:Script>
```

```
    <mx:states>
        <mx:State name="up">
        </mx:State>
        <mx:State name="over">
            <mx:SetProperty target="{this}"
                name="rectFill" value="0x00CC33"/>
            <mx:SetProperty target="{myFilter}"
                name="distance" value="4"/>
        </mx:State>
        <mx:State name="down">
            <mx:SetProperty target="{this}"
                name="rectFill" value="0x00CC33"/>
            <mx:SetProperty target="{myFilter}"
                name="inner" value="true"/>
            <mx:SetProperty target="{myFilter}"
                name="distance" value="2"/>
        </mx:State>
    </mx:states>
</mx:UIComponent>
```

This examples defines skins the for following states:

up Does not define any changes; therefore, the base view state defines the skin for the up state.

over Changes the fill color to 0x00CC33, sets the line width to 2 pixels, and creates a 2-pixel wide drop shadow.

down Changes the fill color to 0x00CC33, sets the drop shadow type to inner, and creates a 2-pixel wide drop shadow.

The following application uses this skin:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatefulSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|Button {
          skin: ClassReference("myComponents.MyButtonStatefulSkin");
          disabledSkin: ClassReference("mx.skins.halo.ButtonSkin");
          selectedUpSkin: ClassReference("mx.skins.halo.ButtonSkin");
          selectedOverSkin: ClassReference("mx.skins.halo.ButtonSkin");
          selectedDownSkin: ClassReference("mx.skins.halo.ButtonSkin");
          selectedDisabledSkin: ClassReference("mx.skins.halo.ButtonSkin");
        }
    </fx:Style>
    <mx:Button label="Hello" id="b" />
</s:Application>
```

Creating a stateful Halo skin using images

You can use images in a stateful skin where a change of state causes the skin to display a different image. One issue when using images is that the base view state must contain the image so that when the skin is first created it has values for the `measuredWidth` and `measuredHeight` properties, otherwise Flex sets the height and width of the skin to 0.

In the following example, you embed images for the up, over, down, and disabled states of the Button control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- skins/myComponents/MyButtonStatefulSkinImages.mxml -->
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            // Embed the skin images.
            [Bindable]
            [Embed(source="../../assets/orb_up_skin.gif")]
            private var buttonUp:Class;
            [Bindable]
            [Embed(source="../../assets/orb_over_skin.gif")]
            private var buttonOver:Class;
            [Bindable]
            [Embed(source="../../assets/orb_down_skin.gif")]
            private var buttonDown:Class;
            [Bindable]
            [Embed(source="../../assets/orb_disabled_skin.gif")]
            private var buttonDisabled:Class;
        ]]>
    </mx:Script>
    <mx:states>
        <mx:State name="up"/>
        <mx:State name="notBase">
            <mx:RemoveChild target="{baseButton}"/>
        </mx:State>
        <mx:State name="over" basedOn="notBase">
            <mx:AddChild creationPolicy="all">
                <mx:Image source="{buttonOver}"
                    maintainAspectRatio="false"
                    width="100%" height="100%"/>
            </mx:AddChild>
        </mx:State>
        <mx:State name="down" basedOn="notBase">
            <mx:AddChild creationPolicy="all">
                <mx:Image source="{buttonDown}"
                    maintainAspectRatio="false"
                    width="100%" height="100%"/>
            </mx:AddChild>
        </mx:State>
        <mx:State name="disabled" basedOn="notBase">
            <mx:AddChild creationPolicy="all">
                <mx:Image source="{buttonDisabled}"
                    maintainAspectRatio="false"
                    width="100%" height="100%"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:Image id="baseButton"
        width="100%" height="100%"
        source="{buttonUp}"
        maintainAspectRatio="false"/>
</mx:Canvas>
```

In this example the skin performs the following actions:

- Defines no changes from the base view state to create the up view state.

- Defines the notBase view state to remove the image defined by the base view state. All other view states, except for the up view state, are based on the notBase view state.
- Sets the `creationPolicy` property to `all` for each AddChild tag. This property specifies to create the child instance at application startup, rather than on the first change to the view state. This prevents flickering when viewing a view state for the first time. For more information, see “View states” on page 1847.
- Sets the `width` and `height` properties to 100% for the Image tags because the Canvas container is what is being resized by the skin parent, not the individual Image controls. This setting configures the Image control to size itself to its parent.
- Sets the `maintainAspectRatio` property to `false` on each Image tag so that the image stretches to fill the full size of the Canvas container.

Using transitions with a stateful Halo skin

View states let you change appearance of a component, typically in response to a user action. Transitions define how a change of view state looks as it occurs on the screen. You define a transition by using the effect classes, in combination with several effects designed explicitly for handling transitions. For more information on transitions, see “Transitions” on page 1870.

In the following example, you add a transition to the stateful skin definition from the previous section. In this example, the transition defines a 100 ms animation to occur when changing the fill color of the skin:

```
<?xml version="1.0"?>
<!-- skins/myComponents/MyButtonStatefulSkinTrans.mxml -->
<mx:UIComponent xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import flash.filters.DropShadowFilter;

      // Define a drop shadow for the over and down states.
      [Bindable]
      private var myFilter:DropShadowFilter = new DropShadowFilter(0);

      // Define a private var for line weight.
      private var _lineWeight:Number = 1;

      // Define public setter and getter for line weight.
      public function get lineWeight():Number
      {
        return _lineWeight;
      }

      public function set lineWeight(value:Number):void
      {
        _lineWeight = value;
        invalidateDisplayList();
      }

      // Define a private var for the fill color.
      private var _rectFill:uint = 0x00FF00;

      // Define public setter and getter for fill color.
      public function get rectFill():uint
      {
        return _rectFill;
      }

      public function set rectFill(value:uint):void
      {
        _rectFill = value;
        invalidateDisplayList();
      }
      override protected function updateDisplayList(unscaledWidth:Number,
          unscaledHeight:Number):void
      {
        graphics.lineStyle(lineWeight, 0x0066FF);
        graphics.beginFill(rectFill, 0.50);
        graphics.drawRoundRect(0, 0, unscaledWidth, unscaledHeight, 10, 10);
        filters = [myFilter];
      }
    ]]>
  </mx:Script>

    <mx:states>
        <mx:State name="up">
        </mx:State>
        <mx:State name="over">
            <mx:SetProperty target="{this}"
                name="rectFill" value="0x00CC33"/>
            <mx:SetProperty target="{myFilter}"
                name="distance" value="4"/>
        </mx:State>
        <mx:State name="down">
            <mx:SetProperty target="{this}"
                name="rectFill" value="0x00CC33"/>
            <mx:SetProperty target="{myFilter}"
                name="inner" value="true"/>
            <mx:SetProperty target="{myFilter}"
                name="distance" value="2"/>
        </mx:State>
    </mx:states>
    <mx:transitions>
        <mx:Transition>
            <mx:AnimateProperty target="{this}"
                property="rectFill" duration="100"/>
        </mx:Transition>
    </mx:transitions>
</mx:UIComponent>
```

Creating advanced programmatic skins for MX components

Accessing the parent component

It is possible to get a reference to the parent of the programmatic skin from within the programmatic skin class. You can use this reference to access properties of the parent component or call methods on it.

You can access the parent from the updateDisplayList() method by using the skin's `parent` property. You cannot access the parent in the skin's constructor because the skin has not yet been added to the parent control. The value of the skin's `parent` property is set when the parent MX container calls the `addChild()` method to add the skin as a child.

When instantiating components with programmatic skins, the order of events is as follows:

1 Create an instance of the parent component.

2 Create an instance of the skin class.

3 Call the `addChild()` method on the parent MX container to add the skin class.

To get a reference to the skin's parent, you must cast the skin's `parent` property to a UIComponent. The skin inherits this read-only property from the IFlexDisplayObject interface. You should also confirm that the parent is a UIComponent by using the `is` operator, because Flex throws a run-time error if the cast cannot be made.

The following example gets the class name of the parent control and draws the border and fill, depending on the type of component the parent is:

```
package {
import flash.display.GradientType;
import flash.display.Graphics;
import mx.skins.Border;
import mx.styles.StyleManager;
import mx.utils.ColorUtil;
import mx.skins.halo.HaloColors;
import mx.core.UIComponent;
public class IconSkin extends Border {
  public function IconSkin() {
     //super();
  }

  override public function get measuredWidth():Number {
     return 14;
  }
  override public function get measuredHeight():Number {
     return 14;
  }

  override protected function updateDisplayList(w:Number, h:Number):void {
     super.updateDisplayList(w, h);
     // User-defined styles
     var borderColor:uint = getStyle("borderColor");
     var fillAlphas:Array = getStyle("fillAlphas");
     var fillColors:Array = getStyle("fillColors");
     styleManager.getColorNames(fillColors);
     var highlightAlphas:Array = getStyle("highlightAlphas");
     var themeColor:uint = getStyle("themeColor");

     var r:Number = width / 2;

     var upFillColors:Array;
     var upFillAlphas:Array;
     var disFillColors:Array;
     var disFillAlphas:Array;
     var g:Graphics = graphics;
     g.clear();

     var myParent:String;

     switch (name) {
        case "upIcon": {
           upFillColors = [ fillColors[0], fillColors[1] ];
           upFillAlphas = [ fillAlphas[0], fillAlphas[1] ];
           if (parent is UIComponent) {
            myParent = String(UIComponent(parent).className);
           }
           if (myParent=="RadioButton") {
            // RadioButton border
            g.beginGradientFill(GradientType.LINEAR,
                [ borderColor, 0x000000 ],
                [100,100], [0,0xFF],
                verticalGradientMatrix(0,0,w,h));
            g.drawCircle(r,r,r);
            g.drawCircle(r,r,(r-1));
            g.endFill();
            // RadioButton fill
            g.beginGradientFill(GradientType.LINEAR,
                upFillColors,
                upFillAlphas,
                [0,0xFF],
                verticalGradientMatrix(1,1,w-2,h-2));
            g.drawCircle(r,r,(r-1));
            g.endFill();
           } else if (myParent=="CheckBox") {
            // CheckBox border
            drawRoundRect(0,0,w,h,0,
                [borderColor, 0x000000], 1,
                verticalGradientMatrix(0,0,w,h),
                GradientType.LINEAR,
                null, {x: 1,y:1,w:w-2,h:h-2,r:0});
            // CheckBox fill
            drawRoundRect(1, 1, w-2, h-2, 0,
                upFillColors, upFillAlphas,
                verticalGradientMatrix(1,1,w-2,h-2));
           }
           // top highlight
           drawRoundRect(1, 1, w-2,
                (h-2)/2, {tl:r,tr:r,bl:0,br:0},
                [0xFFFFFF, 0xFFFFFF],
                highlightAlphas,
                verticalGradientMatrix(0,0,w-2,(h-2)/2));
        }

        // Insert other cases such as downIcon and overIcon here.

     }
  }
}
}
```

Making properties styleable

In many cases, you define a programmatic skin that defines style properties, such as the background color of the skin, the border thickness, or the roundness of the corners. You can make these properties styleable so that your users can change their values in a CSS file or with the `setStyle()` method from inside their applications. You cannot set styles that are defined in programmatic skins by using inline syntax.

To make a custom property styleable, add a call to the `getStyle()` method in the `updateDisplayList()` method and specify that property as the method's argument. When Flex renders the skin, it calls `getStyle()` on that property to find a setting in CSS or on the display list. You can then use the value of the style property when drawing the skin.

You should wrap this call to the `getStyle()` method in a check to see if the style exists. If the property was not set, the result of the `getStyle()` method can be unpredictable.

The following example verifies if the property is defined before assigning it a value:

```
if (getStyle("lineThickness")) {
    _lineThickness = getStyle("lineThickness");
}
```

You must define a default value for the skin's styleable properties. You usually do this in the skin's constructor function. If you do not define a default value, the style property is set to `NaN` or `undefined` if the application does not define that style. This can cause a run-time error.

The following example of the MyButtonSkin programmatic skin class defines default values for the `_lineThickness` and `_backgroundFillColor` styleable properties in the skin's constructor. It then adds calls to the `getStyle()` method in the `updateDisplayList()` method to make these properties styleable:

```
package { // Use unnamed package if this skin is not in its own package.
  // skins/ButtonStylesSkin.as
  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  public class ButtonStylesSkin extends ProgrammaticSkin {
     public var _backgroundFillColor:Number;
     public var _lineThickness:Number;
     // Constructor.
     public function ButtonStylesSkin() {
        // Set default values.
        _backgroundFillColor = 0xFFFFFF;
        _lineThickness=2;
     }
     override protected function updateDisplayList(w:Number, h:Number):void {
        if (getStyle("lineThickness")) {
           // Get value of lineThickness style property.
           _lineThickness = getStyle("lineThickness");
        }
        if (getStyle("backgroundFillColor")) {
           // Get value of backgroundFillColor style property.
           _backgroundFillColor = getStyle("backgroundFillColor");
        }
        // Draw the box using the new values.
        var g:Graphics = graphics;
        g.clear();
        g.beginFill(_backgroundFillColor,1.0);
        g.lineStyle(_lineThickness, 0xFF0000);
        g.drawRect(0, 0, w, h);
        g.endFill();
        g.moveTo(0, 0);
        g.lineTo(w, h);
        g.moveTo(0, h);
        g.lineTo(w, 0);
     }
  }
} // Close unnamed package.
```

In your application, you can set the values of styleable properties by using CSS or the `setStyle()` method.

The following example sets the value of styleable properties on all Button controls with CSS:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStylesSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600" height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";
     mx|Button {
        upSkin:ClassReference('ButtonStylesSkin');
        downSkin:ClassReference('ButtonStylesSkin');
        overSkin:ClassReference('ButtonStylesSkin');
        disabledSkin:ClassReference('ButtonStylesSkin');
        lineThickness:4;
        backgroundFillColor:#CCCCCC;
     }
  </fx:Style>
  <fx:Script><![CDATA[
     public function changeLineThickness(e:Event):void {
        var t:int = Number(b1.getStyle("lineThickness"));
        if (t == 4) {
           b1.setStyle("lineThickness",1);
        } else {
           b1.setStyle("lineThickness",4);
        }
     }
  ]]></fx:Script>
  <mx:Button id="b1" label="Change Line Thickness" click="changeLineThickness(event)"/>

</s:Application>
```

When using the `setStyle()` method to set the value of a style property in your application, you can set the value of a styleable property on a single component instance (as in the previous example) or on all instances of a component.

The following example uses the `setStyle()` method to set the value of a styleable property on all instances of the control (in this case, all MX Button controls):

```
<?xml version="1.0"?>
<!-- skins/ApplyGlobalButtonStylesSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600" height="600">
  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|Button {
        upSkin:ClassReference('ButtonStylesSkin');
        downSkin:ClassReference('ButtonStylesSkin');
        overSkin:ClassReference('ButtonStylesSkin');
        disabledSkin:ClassReference('ButtonStylesSkin');
        lineThickness:4;
        backgroundFillColor:#CCCCCC;
     }
  </fx:Style>
  <fx:Script><![CDATA[
     public function changeLineThickness(e:Event):void {
        var t:int = Number(b1.getStyle("lineThickness"));
        if (t == 4) {
         styleManager.getStyleDeclaration("mx.controls.Button").setStyle("lineThickness", 1);
        } else {
         styleManager.getStyleDeclaration("mx.controls.Button").setStyle("lineThickness", 4);
        }
     }
  ]]></fx:Script>
  <mx:Button id="b1" label="Change Line Thickness" click="changeLineThickness(event)"/>

</s:Application>
```

If you do not set the values of these properties using either CSS or the `setStyle()` method, Flex uses the default values of the properties that you set in the skin's constructor.

To get the value of an existing style property, such as `color` or `fontSize`, you do not have to wrap the call to the `getStyle()` method in a check for the property's existence. This is because Flex creates a CSSStyleDeclaration that defines the default values of all of a component's styles. Preexisting style properties are never undefined. Style properties that you add to a custom skin are not added to this CSSStyleDeclaration because the component does not know that the property is a style property.

Custom skin properties that you define as styleable are noninheritable. So, subclasses or children of that component do not inherit the value of that property.

Embedding assets

Many applications built in Adobe® Flex™ use external assets like images, sounds, and fonts. Although you can reference and load assets at run time, you often compile these assets into your applications. The process of compiling an asset into your application is called *embeddingthe asset*. Flex lets you embed image files, movie files, MP3 files, and TrueType fonts into your applications.

For information on embedding fonts, see "Fonts" on page 1568.

About embedding assets

When you embed an asset, you compile it into your application's SWF file. The advantage of embedding an asset is that it is included in the SWF file, and can be accessed faster than it can if the application has to load it from a remote location at run time. The disadvantage of embedding an asset is that your SWF file is larger than if you load the asset at run time.

You should use the Image tag when embedding a general purpose image file in an application. The Spark component set also includes a BitmapImage class, but that class should be used only for embedding images in skins and FXG components. For more information, see "Image control" on page 676.

Examples of embedding assets

One of the most common uses of the embed mechanism is to import an image for a Flex control by using the `@Embed()` directive in an MXML tag definition. For example, many controls support icons or skins that you can embed in the application. The Button control lets you specify label text, as well as an optional icon image, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed\ButtonIcon.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Button label="Icon Button"
        height="30"
        icon="@Embed(source='logo.gif')"/>
</s:Application>
```

Another option for embedding is to associate the embedded image with a variable by using the `[Embed]` metadata tag. In this way, you can reference the embedded image from multiple locations in your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed\ButtonIconClass.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            [Embed(source="logo.gif")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </fx:Script>
    <s:Button label="Icon Button 1"
        height="30"
        icon="{imgCls}"/>
    <s:Button label="Icon Button 2"
        height="30"
        icon="{imgCls}"/>
</s:Application>
```

For style properties, you can embed an asset as part of a style sheet definition by using the `Embed()` directive, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed\ButtonIconCSS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Style>
        .myCustomButton {
            icon:Embed(source="logo.gif");
        }
    </fx:Style>
    <s:Button label="Icon Button Style Def"
        styleName="myCustomButton"/>
</s:Application>
```

Note: *The equal sign (=) in the style sheet is a Flex extension that may not be supported by all CSS processors. If you find that it is not supported, you can use the* `Embed(filename)` *syntax.*

Accessing assets at run time

The alternative to embedding an asset is to load the asset at run time. You can load an asset from the local file system in which the SWF file runs, or you can access a remote asset, typically through an HTTP request over a network.

Embedded assets load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded assets also require you to recompile your applications whenever your asset changes.

Assets loaded at run time exist as separate, independent files on your web server (or elsewhere) and are not compiled into your Flex applications. The referenced assets add no overhead to an application's initial load time. However, you might experience a delay when you use the asset and load it in Adobe® Flash® Player or Adobe® AIR™. These assets are independent of your Flex application, so you can change them without causing a recompile operation, as long as the names of the modified assets remain the same.

For examples that load an asset at run time, see "Image control" on page 676and "SWFLoader control" on page 715.

For security, by default Flash Player does not allow an application to access some types of remote data (such as SWF files) at run time from a domain other than the domain from which the application was served. Therefore, a server that hosts data must be in the same domain as the server hosting your application, or the server must define a crossdomain.xml file. A crossdomain.xml file is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from specific domains, or from all domains. For more information on application security, see "Security" on page 117.

Supported file types

You can embed the following types of files in a Flex application.

File Type	File Format	MIME Type	Description and Examples
Images	GIF	image/gif	Embedding JPEG, GIF, and PNG images
Images	.JPG, .JPEG	image/jpeg	Embedding JPEG, GIF, and PNG images
Images	PNG	image/png	Embedding JPEG, GIF, and PNG images
Images	SVG	image/svg image/svg-xml	Embedding SVG images
Flash	SWF	application/x-shockwave-flash	Embedding SWF files

File Type	File Format	MIME Type	Description and Examples
Flash	Symbols stored in a SWF file	application/x-shockwave-flash	Embedding SWF files
Audio	MP3	audio/mpeg	Embedding sounds
Font	TTF (TrueType)	application/x-font-truetype	
Font	FON (System font)	application/x-font	
All other types		application/octet-stream	Embedding all other file types

Syntax for embedding assets

The syntax that you use for embedding assets depends on where in your application you embed the asset. Flex supports the following syntaxes:

- `[Embed(`*parameter1, paramater2, ...*`)]` metadata tag

  You use this syntax to embed an asset in an ActionScript file, or in an `<fx:Script>` block in an MXML file. For more information, see "Using the [Embed] metadata tag" on page 1705.

- `@Embed(`*parameter1, paramater2, ...*`)` directive

  You use this syntax in an MXML tag definition to embed an asset. For more information, see "Using the @Embed() directive in MXML" on page 1705.

- `Embed(`*parameter1, paramater2, ...*`)` directive

  You use this syntax in an `<fx:Style>` block in an MXML file to embed an asset. For more information, see "Embedding assets in style sheets" on page 1706.

All three varieties of the embed syntax let you access the same assets; the only difference is where you use them in your application.

Escaping the @ character

You can use the slash character (\) to escape the at sign character (@) when you want to use a literal @ character. For example, the string "\@Embed(foo)" means the literal string "@Embed(foo)"). You use two slash characters (\\) to escape a single backslash character. For example, use the character string "\@" to specify the literal strings "\@".

Embed parameters

Each form of the embed syntax takes one or more optional parameters. The exact syntax that you use to embed assets depends on where they are embedded. Some of these parameters are available regardless of what type of asset you are embedding, and others are specific to a particular type of media. For example, you can use the `source` and `mimeType` parameters with any type of media, but the `scaleGridRight` parameter applies only to images.

The following table describes the parameters that are available for any type of embedded asset. For more information, see "About the source parameter" on page 1703 and "About the MIME type" on page 1704.

Parameter	Description
`source=`*`asset`*	Specifies the name and path of the asset to embed; either an absolute path or a path relative to the file containing the embed statement. The embedded asset must be a locally stored asset. Therefore you cannot specify a URL for an asset to embed. For more information on setting the path, see “About setting the path to the embedded asset” on page 1704.
`mimeType=`*`type`*	Specifies the mime type of the asset. For more information, see “About the MIME type” on page 1704.
`smoothing=true|false`	Specifies whether the bitmap is smoothed when scaled. If `true`, the bitmap is smoothed when scaled. If `false`, the bitmap is not smoothed when scaled. This property can be used with any image format.
`compression=true|false`	For lossless images only (GIF and PNG) specifies to compress the embedded image in the generated SWF file.
`quality=`***`val`***	If you specify `compression=true`, specifies a percentage value, between 0 and 100, to control the compression algorithm. The lower the value, the higher the amount of compression.

The following table describes the parameters that are specific for images and Sprite objects. For more information, see “Using 9-slice scaling with embedded images” on page 1711.

Parameter	Description
`scaleGridTop`	Specifies the distance in pixels of the upper dividing line from the top of the image in a 9-slice scaling formatting system. The distance is relative to the original, unscaled size of the image.
`scaleGridBottom`	Specifies the distance in pixels of the lower dividing line from the top of the image in a 9-slice scaling formatting system. The distance is relative to the original, unscaled size of the image.
`scaleGridLeft`	Specifies the distance in pixels of the left dividing line from the left side of the image in a 9-slice scaling formatting system. The distance is relative to the original, unscaled size of the image.
`scaleGridRight`	Specifies the distance in pixels of the right dividing line from the left side of the image in a 9-slice scaling formatting system. The distance is relative to the original, unscaled size of the image.

The following table describes the parameter that is specific to SWF files. For more information, see “Embedding SWF files” on page 1709.

Parameter	Description
`symbol`	Specifies the symbol in a SWF file to embed, for use with Adobe Flash Player 8 and earlier, and for Flash Player 9 static assets. Use static assets for simple artwork or skins that do not contain any ActionScript 3.0 code.

About the source parameter

In almost all cases, you must specify the `source` parameter, or nothing is embedded.

The `source` parameter is the default parameter of the `[Embed]` metadata tag; therefore, if you are not specifying any other parameters, you can just supply its value without explicitly including the parameter name or assigning it the desired value, as the following example shows:

```
<fx:Style>
    .myCustomButton1 {
        icon:Embed("logo.gif");
    }
    .myCustomButton2 {
        icon:Embed(source="logo.gif");
    }
</fx:Style>
```

About setting the path to the embedded asset

You can specify a fully qualified path to the image, as the following examples show:

```
<s:Button label="Icon Button"
    icon="@Embed(source='c:/myapp/assets/logo.gif')"/>
```

Note: *Do not use the backslash character (\) as a separator in the path.*

If the path does not start with a slash character, Flex first searches for the file relative to the file that contains the `[Embed]` metadata tag. For example, the MXML file testEmbed.mxml includes the following code:

```
<s:Button label="Icon Button" icon="@Embed(source='assets/logo.gif')"/>
```

In this example, Flex searches the subdirectory named assets in the directory that contains the testEmbed.mxml file. If the image is not found, Flex then searches for the image in the SWC files associated with the application.

If the path starts with a slash character, Flex first searches the directory of the MXML file for the asset, and then it searches the source path. You specify the source path to the Flex compiler by using the `source-path` compiler option. For example, you set the `source-path` option as the following code shows:

```
-source-path=a1,a2,a3
```

The MXML file a1/testEmbed.mxml then uses the following code:

```
<s:Button label="Icon Button" icon="@Embed(source='/assets/logo.gif')"/>
```

Flex first searches for the file in a1/assets, then in a2/assets, and then in a3/assets. If the image is not found, Flex searches for the image in the SWC files associated with the application.

If the MXML file is in the a2 directory, as in a2/testEmbed.mxml, Flex first searches the a2 directory and then the directories specified by the `source-path` option.

About the MIME type

You can optionally specify a MIME type for the imported asset by using the `mimeType` parameter. If you do not specify a `mimeType` parameter, Flex makes a best guess about the type of the imported file based on the file extension. If you do specify it, the `mimeType` parameter overrides the default guess of the asset type.

Flex supports the following MIME types.

- application/octet-stream
- application/x-font
- application/x-font-truetype
- application/x-shockwave-flash
- audio/mpeg
- image/gif
- image/jpeg

- image/png
- image/svg
- image/svg-xml

Using the [Embed] metadata tag

You can use the `[Embed]` metadata tag to import JPEG, GIF, PNG, SVG, SWF, TTF, and MP3 files.

You must use the `[Embed]` metadata tag before a variable definition, where the variable is of type Class. The following example loads an image file, assigns it to the `imgCls` variable, and then uses that variable to set the value of the `source` property of an Image control:

```
<?xml version="1.0"?>
<!-- embed\ImageClass.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100" height="80">

    <fx:Script>
        <![CDATA[
            [Embed(source="logo.gif")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </fx:Script>
    <s:Image source="{imgCls}"/>
</s:Application>
```

Notice that Flex uses data binding to tie the `imgCls` variable to the `source` property. If you omit the `[Bindable]` metadata tag preceding the `imgCls` variable definition, Flex can perform the data binding operation only once, at application startup. When you include the `[Bindable]` metadata tag, Flex recognizes any changes to the `imgCls` variable and updates any components that use that variable when a change to it occurs.

Generally, this method of embedding assets provides more flexibility than other methods because you can import an asset once and then use it in multiple places in your application, and because you can update the asset and have the data binding mechanism propagate that update throughout your application.

Using the @Embed() directive in MXML

Many Flex components, such as Button and TabNavigator, take an `icon` property or other property that lets you specify an image to the control. You can embed the image asset by specifying the property value by using the `@Embed()` directive in MXML. You can use any supported graphic file with the `@Embed()` directive, including SWF files and assets inside SWF files.

You can use the `@Embed()` directive to set an MXML tag property, or to set a property value by using a child tag. The `@Embed()` directive returns a value of type Class or String. If the component's property is of type String, the `@Embed()` directive returns a String. If the component's property is of type Class, the `@Embed()` directive returns a Class. Using the `@Embed()` directive with a property that requires a value of any other data type results in an error.

The following example creates a Button control and sets its `icon` property by using the `@Embed()` directive:

```
<?xml version="1.0"?>
<!-- embed\ButtonAtEmbed.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Button label="Icon Button"
        height="30"
        icon="@Embed(source='logo.gif')"/>

</s:Application>
```

Embedding assets in style sheets

Many style properties of Flex components support imported assets. The following examples show different ways to embed assets in style sheets:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- embed/EmbedWithStyleSheet.mxml -->
<s:Application name="Spark_Button_icon_test"
        xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout horizontalAlign="center" verticalAlign="middle"/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            [Embed("logo.gif")]
            protected const ICON:Class;
        ]]>
    </fx:Script>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";

        s|Button#btn1 {
            icon: "logo.gif";
        }

        s|Button#btn2 {
            icon: Embed("logo.gif");
        }

        s|Button#btn3 {
            icon: PropertyReference("ICON");
        }
    </fx:Style>

    <s:Button id="btn1" label="Spark Button with dynamic icon"/>
    <s:Button id="btn2" label="Spark Button with inline embedded icon"/>
    <s:Button id="btn3" label="Spark Button with metadata [Embed] icon"/>

</s:Application>
```

This example is used with permission by http://blog.flexexamples.com.

You can also use these style properties to set the skins for a component. *Skinning* is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of images, SWF files, or class files that contain drawing API methods.

For more information on skinning and on embedding assets by using style sheets, see “Skinning MX components” on page 1655.

Embedding asset types

You can import various types of media, including images, SWF files, and sound files.

Embedding JPEG, GIF, and PNG images

Flex supports embedding JPEG, GIF, and PNG files. You can use these images for icons, skins, and other types of application assets.

You might want to manipulate an embedded image at run time. To manipulate it, you determine the data type of the object representing the image and then use the appropriate ActionScript methods and properties of the object.

For example, you use the `[Embed]` metadata tag in ActionScript to embed a GIF image, as the following code shows:

```
[Embed(source="logo.gif")]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded image. When embedding JPEG, GIF, and PNG files, Flex defines imgCls as a reference to a subclass of the mx.core.BitmapAsset class, which is a subclass of the flash.display.Bitmap class.

In ActionScript, you can create and manipulate an object that represents the embedded image before passing it to a control. To do so, you create an object with the type of the embedded class, manipulate it, and then pass the object to the control, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed/EmbedAccessClassObject.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.core.BitmapAsset;

            [Embed(source="logo.gif")]
            [Bindable]
            public var imgCls:Class;
            private function modImage():void {
                // Create an object from the embed class.
                // Since the embedded image is a GIF file,
                // the data type is BitmapAsset.
                var imgObj:BitmapAsset = new imgCls() as BitmapAsset;

                // Modify the object.
                imgObj.bitmapData.noise(4);
                // Write the modified object to the Image control.
                myImage.source=imgObj;
            }
        ]]>
    </fx:Script>

    <s:HGroup>
        <s:Image id="myImageRaw" source="{imgCls}"/>
        <s:Image id="myImage" creationComplete="modImage();"/>
    </s:HGroup>
</s:Application>
```

In this example, the first Image control displays the unaltered image and the second Image control displays the modified image. You use the `bitmapData` property of the mx.core.BitmapAsset class to modify the object. The `bitmapData` property is of type flash.display.BitmapData; therefore you can use all of the methods and properties of the BitmapData class to manipulate the object.

Embedding SVG images

Flex supports importing Scalable Vector Graphics (SVG) images, or a GZip compressed SVG image in a SVGZ file, into an application. This lets you import SVG images and use SVG images as icons for Flex controls.

Flex supports a subset of the SVG 1.1 specification to let you import static, two-dimensional scalable vector graphics. This includes support for basic SVG document structure, Cascading Style Sheets (CSS) styling, transformations, paths, basic shapes, colors, and a subset of text, painting, gradients, and fonts. Flex does not support SVG animation, scripting, or interactivity with the imported SVG image.

For example, you use the `[Embed]` metadata tag in ActionScript to embed an SVG image, as the following code shows:

```
[Embed(source="logo.svg")]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded image. When embedding an SVG image, Flex defines imgCls as a reference to a subclass of the mx.core.SpriteAsset class, which is a subclass of the flash.display.Sprite class. Therefore, you can manipulate the image by using the methods and properties of the SpriteAsset class. For an example that manipulates an imported image, see "Embedding JPEG, GIF, and PNG images" on page 1707.

Embedding sounds

Flex supports embedding MP3 sound files for later playback. The following example creates a simple media player with Play and Stop buttons:

```
<?xml version="1.0"?>
<!-- embed/EmbedSound.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[

            import flash.media.*;

            [Embed(source="sample.mp3")]
            [Bindable]
            public var sndCls:Class;

            public var snd:Sound = new sndCls() as Sound;
            public var sndChannel:SoundChannel;

            public function playSound():void {
                sndChannel=snd.play();
            }

            public function stopSound():void {
                sndChannel.stop();
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:Button label="play" click="playSound();"/>
        <s:Button label="stop" click="stopSound();"/>
    </s:HGroup>
</s:Application>
```

In this example, you define a class named sndCls that represents the embedded MP3 file. When embedding MP3 files, Flex defines sndCls as a reference to a subclass of the mx.core.SoundAsset, which is a subclass of the flash.media.Sound class.

Flex can handle any legal filename, including filenames that contain spaces and punctuation marks. If the MP3 filename includes regular quotation marks, be sure to use single quotation marks around the filename.

You do not have to embed the sound file to use it with Flex. You can also use the Sound class to load a sound file at run time. For more information, see the Sound class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Embedding SWF files

Flex fully supports embedding Flash SWF files. You can embed different types of SWF files.

Embedding SWF files for Flash Player 8 and earlier, and for static Flash Player 9 or later assets

You can embed SWF files created for Flash Player 8 and earlier and for Flash Player 9 or later static assets. Use static assets for simple artwork or skins that do not contain any ActionScript 3.0 code. When embedded, your Flex application cannot interact with the embedded SWF file. That is, you cannot use ActionScript in a Flex application to access the properties or methods of a SWF file created for Flash Player 8 or earlier or for static Flash Player 9 or later assets.

***Note:** You can use the flash.net.LocalConnection class to communicate between a Flex application and a SWF file.*

For example, you use the `[Embed]` metadata tag in ActionScript to embed a SWF file, as the following code shows:

```
[Embed(source="icon.swf")]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded SWF file. Flex defines imgCls as a reference to a subclass of the mx.core.MovieClipAsset class, which is a subclass of the flash.display.MovieClip class. Therefore you can manipulate the image by using the methods and properties of the MovieClipAsset class.

Embedding SWF symbols

Flex lets you reference exported symbols in an embedded SWF file. If the symbol has dependencies, Flex embeds them also; otherwise, Flex embeds only the specified symbol from the SWF file. To reference a symbol, you specify the `symbol` parameter:

```
[Embed(source='SWFFileName.swf', symbol='symbolName')]
```

***Note:** Flash defines three types of symbols: Button, MovieClip, and Graphic. You can embed Button and MovieClip symbols in a Flex application, but you cannot embed a Graphic symbol because it cannot be exported for ActionScript.*

This capability is useful when you have a SWF file that contains multiple exported symbols, but you want to load only some of them into your Flex application. Loading only the symbols that your application requires makes the resulting Flex SWF file smaller than if you imported the entire SWF file.

A Flex application can import any number of SWF files. However, if two SWF files have the same filename and the exported symbol names are the same, you cannot reference the duplicate symbols, even if the SWF files are in separate directories.

If the SWF file contains any ActionScript code, Flex prints a warning during compilation and then strips out the ActionScript from the embed symbol. This means that you can only embed the symbol itself.

The following example imports a green square from a SWF file that contains a library of different shapes:

```
<?xml version="1.0"?>
<!-- embed\EmbedSWFSymbol.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
    <s:Image id="image0"
        source="@Embed(source='circleSquare.swf', symbol='greenSquare')"/>
</s:Application>
```

If you use the `[Embed]` metadata tag in ActionScript to embed the symbol, you can access the object that represents the symbol, as the following code shows:

```
[Embed(source='shapes.swf', symbol='greenSquare')]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded symbol. Internally, Flex defines imgCls as a reference to a subclass of either one of the following classes:

SpriteAsset For single-frame SWF files

MovieClipAsset For multiframe SWF files

Embedding SWF files that represent Flex applications

You can embed SWF files that represent Flex applications. For example, you use the `[Embed]` metadata tag in ActionScript to embed a SWF file, as the following code shows:

```
[Embed(source="flex2.swf")]
[Bindable]
public var flexAppCls:Class;
```

In this example, you define a class named flexAppCls that represents the embedded SWF file. Flex defines flexAppCls as a reference to a subclass of the mx.core.MovieClipAsset class, which is a subclass of the flash.display.MovieClip class. Therefore you can manipulate the embedded SWF file by using the methods and properties of the MovieClipAsset class.

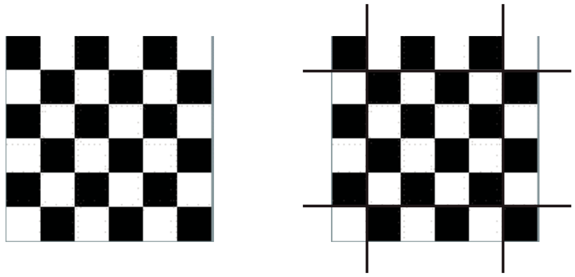
You typically embed a Flex application when you do not require the embedding application to interact with the embedded application. If the embedding application requires interactivity with the embedded application, you might consider implementing it as a custom component, rather than as a separate application.

Alternatively, if you use the SWFLoader control to load the Flex application at run time, the embedding application can interact with the loaded application to access its properties and methods. For more information and examples, see "Interacting with a loaded Flex application" on page 717.

Using 9-slice scaling with embedded images

Flex supports the 9-slice scaling of embedded images. This feature lets you define nine sections of an image that scale independently. The nine regions are defined by two horizontal lines and two vertical lines running through the image, which form the inside edges of a 3 by 3 grid. For images with borders or fancy corners, 9-slice scaling provides more flexibility than full-graphic scaling.

The following example show an image, and the same image with the regions defined by the 9-slice scaling borders:



When you scale an embedded image that uses 9-slice scaling, all text and gradients are scaled normally. However, for other types of objects the following rules apply:

- Content in the center region is scaled normally.
- Content in the corners is not scaled.
- Content in the top and bottom regions is scaled only horizontally. Content in the left and right regions is scaled only vertically.
- All fills (including bitmaps, video, and gradients) are stretched to fit their shapes.

If you rotate the image, all subsequent scaling is normal, as if you did not define any 9-slice scaling.

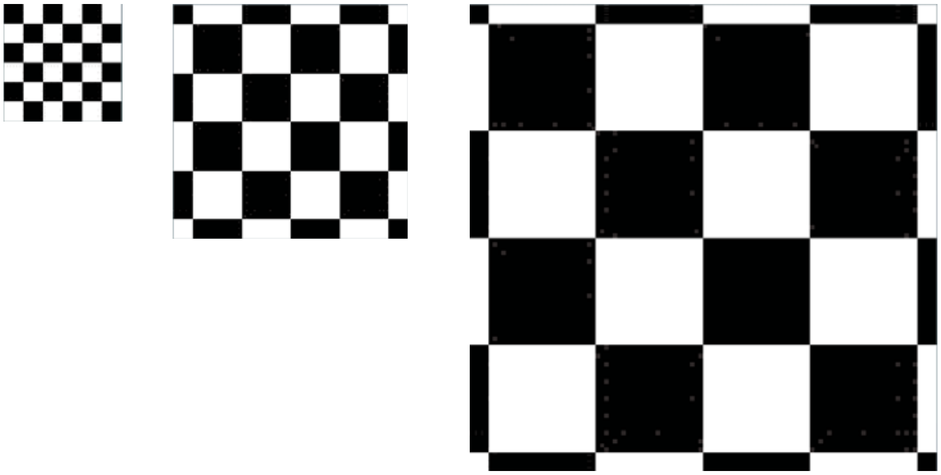
To use 9-slice scaling, define the following four parameters in your embed statement: `scaleGridTop`, `scaleGridBottom`, `scaleGridLeft`, and `scaleGridRight`. For more information on these parameters, see “Embed parameters” on page 1702.

An embedded SWF file may already contain 9-slice scaling information specified by using Adobe Flash Professional. In that case, the SWF file ignores any 9-slice scaling parameters that you specify in the embed statement.

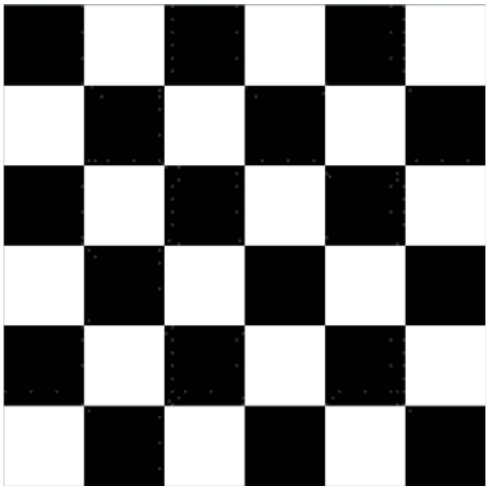
The following example uses 9-slice scaling to maintain a set border, regardless of how the image itself is resized.

```
<?xml version="1.0"?>
<!-- embed\Embed9slice.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="1200" width="600">
    <fx:Script>
        <![CDATA[
            [Embed(source="slice_9_grid.gif",
                scaleGridTop="25", scaleGridBottom="125",
                scaleGridLeft="25", scaleGridRight="125")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </fx:Script>

    <s:VGroup>
        <s:Image source="{imgCls}"/>
        <s:Image source="{imgCls}" width="300" height="300"/>
        <s:Image source="{imgCls}" width="450" height="450"/>
    </s:VGroup>
</s:Application>
```

The original image is 30 by 30 pixels. The preceding code produces a resizable image that maintains a 5-pixel border:



If you had omitted the 9-slice scaling, the scaled image would have appeared exactly like the unscaled image, as the following image shows:



In this example, you define a class named imgCls that represents the embedded image. If the image is a SWF file that uses 9-slice scaling, Flex defines imgCls as a subclass of the mx.core.SpriteAsset class, which is a subclass of the flash.display.Sprite class. Therefore you can use all of the methods and properties of the SpriteAsset class to manipulate the object.

Embedding all other file types

You can embed any file type in a Flex application as a bit map array. However, Flex does not recognize or process files other than those described previously. If you embed any other file type, you must provide the transcode logic to properly present it to the application user.

To load a file type that is not specifically supported by Flex, use the `[Embed]` metadata tag to define the source file and MIME type "application/octet-stream". Then define a new class variable for the embedded file. The embedded object is a ByteArrayAsset type object.

The following code embeds a bitmap (.bmp) file and displays properties of the embedded object when you click the Show Properties button. To display or use the object in any other way, you must provide code to create a supported object type.

```
<?xml version="1.0"?>
<!-- embed\EmbedOtherFileTypes.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.core.UIComponent;
            import mx.core.BitmapAsset;

            [Embed(source="logo.bmp",mimeType="application/octet-stream")]
            private var FileClass : Class;

            private function captureEmbeddedImage():void {
                var fileByteArray:Object = new FileClass();
                myTA.text+= "File length:  " + String(fileByteArray.length) + "\n";
                myTA.text+="File type:  " +
                    String(getQualifiedSuperclassName(fileByteArray)) + "\n";
            }
        ]]>
    </fx:Script>

    <s:Button
        id="showProperties"
        label="Show Properties"
        click="captureEmbeddedImage();"/>

    <s:TextArea id="myTA"
        width="75%"
        height="100"/>
</s:Application>
```

FXG and MXML graphics

Adding graphics to your applications can make them more attractive and usable. In many cases, you might want to add graphics that are vector-based, and not import images that don’t scale well. You can create vector based graphics in Flex by using one of the following APIs:

- FXG
- MXML graphics

FXG is a declarative syntax for defining static graphics. You typically use a graphics tool such as Adobe Illustrator to export an FXG document, and then use the FXG document as an optimized component in your application. FXG graphics are used by the mobile skin classes because they are so lightweight.

MXML graphics, on the other hand, are a collection of classes that you use to define interactive graphics. You typically write MXML graphics code in Flash Builder. You can add interactivity to MXML graphics code because they use classes in the Flex SDK that are subclasses of GraphicElement. The result is not as optimized as FXG.

FXG and MXML graphics define the following:

- Graphics and text primitives
- Fills, strokes, gradients, and bitmaps
- Support for effects such as filters, masks, alphas, transforms, and blend modes

FXG and MXML graphics share very similar syntax. For example, in FXG, you can define a rectangle with the `<Rect>` tag. In MXML graphics, you use the `<s:Rect>` tag. Most FXG elements have MXML graphics equivalents, although the attributes supported on FXG elements are only a subset of those supported on MXML graphics tags.

The amount of interactivity when using FXG and MXML graphics is different. If you use MXML graphics, the tags are mapped to their backing ActionScript implementations during compilation. You can reference the MXML graphic elements and have greater control over them in your code. If you use FXG, then you cannot reference instances of objects within the FXG document from the application or other components. In addition, you cannot add MXML code to it, nor can you add ActionScript.

FXG and MXML graphics do not share the same namespace. MXML graphics use the MXML namespace of the containing document. In most cases, this is the Spark namespace. An FXG document uses its own namespace.

You cannot use fragments of FXG syntax in an MXML file. You must either use the FXG document as a separate component or convert the code to MXML graphics syntax.

You can also draw with the ActionScript drawing APIs, described in Using the drawing API.

More Help topics

Chapter 4 "Graphics" in Flex 4 Cookbook

Graphics classes and elements

FXG and MXML graphics share a very similar syntax. The FXG language includes a set of elements that are defined in the FXG 2.0 specification. MXML graphics include tags that are defined in the MXML language. FXG can be thought of as a subset of the MXML graphics tags.

MXML graphics tags are converted to their backing ActionScript implementations by the MXML compiler. Most FXG tags, on the other hand, are converted to low-level Flash Player instructions when compiled. It is this difference that make FXG components much more highly optimized than MXML graphics.

Most FXG elements are roughly equivalent to an MXML graphics class of the same name. But, they do not support all the properties of the MXML graphics class, nor are they backed by ActionScript classes. The following table shows the FXG 2.0 elements and their supported attributes and children. In addition, it lists the equivalent MXML graphics tag:

FXG tag	**Equivalent MXML tag and ActionScript class**	**FXG-supported attributes**	**FXG children**

`<BitmapFill>`	`<s:BitmapFill>`
	mx.graphics.BitmapFill
		`fillMode`
		`rotation`
		`scaleX`
		`scaleY`
		`source (required)`
		`x`
		`y`
			`<matrix>`

`<BitmapImage>`	`<s:BitmapImage>`
	spark.primitives.BitmapImage
		`alpha`
		`blendMode`
		`fillMode`
		`height`
		`rotation`
		`scaleX`
		`scaleY`
		`source (required)`
		`visible`
		`width`
		`x`
		`y`
			None.

`<ColorTransform>`	`<s:ColorTransform>`
	flash.geom.ColorTransform
		`alphaMultiplier`
		`alphaOffset`
		`blueMultiplier`
		`blueOffset`
		`greenMultiplier`
		`greenOffset`
		`redMultiplier`
		`redOffset`
			None.

`<Definition>`	`<fx:Definition>`	`name (required)`	`<Group>`

`<Ellipse>`	`<s:Ellipse>`
	spark.primitives.Ellipse
		`blendMode`
		`height`
		`rotation`
		`scaleX`
		`scaleY`
		`transformX`
		`transformY`
		`visible`
		`width`
		`x`
		`y`
			`<fill>`
			`<filters>`
			`<mask>`
			`<stroke>`
			`<transform>`

`<GradientEntry>`	`<s:GradientEntry>`
	mx.graphics.GradientEntry
		`alpha`
		`color`
		`ratio`
			None.

`<Graphic>`	`<s:Graphic>`
	spark.primitives.Graphic
		`mask`
		`scaleGridBottom`
		`scaleGridLeft`
		`scaleGridRight`
		`scaleGridTop`
		`version`
		`viewHeight`
		`viewWidth`
			`<Group>`
			`<Library>`
			`<mask>`
			Any subclass of the GraphicElement class

FXG tag	Equivalent MXML tag and ActionScript class	FXG-supported attributes	FXG children

`<Group>`	`<s:Group>`
	spark.components.Group
		alpha
		blendMode
		id
		maskType
		rotation
		scaleGridBottom
		scaleGridLeft
		scaleGridRight
		scaleGridTop
		scaleX
		scaleY
		transformX
		transformY
		visible
		x
		y
			`<filters>`
			`<Group>`
			`<transform>`
			Any subclass of the GraphicElement class

`<Library>`	`<fx:Library>`	None.	`<Definition>`

`<Line>`	`<s:Line>`
	spark.primitives.Line
		alpha
		blendMode
		id
		maskType
		rotation
		scaleX
		scaleY
		transformX
		transformY
		visible
		x
		xFrom
		xTo
		y
		yFrom
		yTo
			`<fill>`
			`<filters>`
			`<mask>`
			`<stroke>`
			`<transform>`

`<LinearGradient>`	`<s:LinearGradient>`
	mx.graphics.LinearGradient
		interpolationMethod
		rotation
		scaleX
		spreadMethod
		x
		y
			`<GradientEntry>`
			`<matrix>`

`<LinearGradientStroke>`	`<s:LinearGradientStroke>`
	mx.graphics.LinearGradientStroke
		caps
		interpolationMethod
		joints
		miterLimit
		rotation
		scaleMode
		scaleX
		spreadMethod
		x
		y
			`<GradientEntry>`
			`<matrix>`

`<Matrix>`	`<s:Matrix>` flash.geom.Matrix	a b c d tx ty	None.

FXG tag	Equivalent MXML tag and ActionScript class	FXG-supported attributes	FXG children

`<Path>`	`<s:Path>`
	spark.primitives.Path
		blendMode
		data
		rotation
		scaleX
		scaleY
		transformX
		transformY
		visible
		winding
		x
		y
			`<fill>`
			`<filters>`
			`<mask>`
			`<stroke>`
			`<transform>`

`<RadialGradient>`	`<s:RadialGradient>`
	mx.graphics.RadialGradient
		focalPointRatio
		interpolationMethod
		rotation
		scaleX
		scaleY
		spreadMethod
		x
		y
			`<GradientEntry>`
			`<matrix>`

`<RadialGradientStroke>`	`<s:RadialGradientStroke>`
	mx.graphics.RadialGradientStroke
		caps
		focalPointRatio
		joints
		miterLimit
		scaleMode
		scaleX
		scaleY
		rotation
		spreadMethod
		interpolationMethod
		x
		y
			`<GradientEntry>`
			`<matrix>`

`<Rect>`	`<s:Rect>`
	spark.primitives.Rect
		blendMode
		bottomLeftRadiusX
		bottomLeftRadiusY
		bottomRightRadiusX
		bottomRightRadiusY
		height
		radiusX
		radiusY
		rotation
		scaleX
		scaleY
		topLeftRadiusX
		topLeftRadiusY
		topRightRadiusX
		topRightRadiusY
		transformX
		transformY
		visible
		width
		x
		y
			`<fill>`
			`<filters>`
			`<mask>`
			`<stroke>`
			`<transform>`

`<SolidColor>`	`<s:SolidColor>` mx.graphics.SolidColor	alpha color	None.

FXG tag	Equivalent MXML tag and ActionScript class	FXG-supported attributes	FXG children

`<SolidColorStroke>`	`<s:SolidColorStroke>`
	mx.graphics.SolidColorStroke
		`alpha`
		`caps`
		`color`
		`joints`
		`miterLimit`
		`pixelHinting`
		`scaleMode`
		`weight`
			None.

`<RichText>`	`<s:RichText>`
	spark.primitives.RichText
		`alpha`
		`blendMode`
		`height`
		`id`
		`maskType`
		`paddingLeft`
		`paddingRight`
		`paddingTop`
		`paddingBottom`
		`rotation`
		`scaleX`
		`scaleY`
		`transformX`
		`transformY`
		`visible`
		`width`
		`x`
		`y`
		Paragraph and character style attributes
			`<br>`<br>`<content>`
			`<p>`
			`<span>`
			`<transform>`

`<Transform>`	`<s:Transform>`
	mx.geom.Transform
		None.	`<colorTransform>`
			`<matrix>`

For complete descriptions of the FXG elements and their supported attributes, see the FXG 2.0 specification.

FXG

FXG is a declarative XML syntax for defining vector graphics in applications built with Flex. FXG can also be used as an interchange format with other Adobe tools such as Illustrator or Photoshop. FXG closely follows the Flash Player 10 rendering model.

Designers can create vector images using tools such as Photoshop, Illustrator and Fireworks and export them as an FXG document. You can then use that FXG document as a component in your applications.

The following example is an FXG document that draws a filled rectangle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/GraphicComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
     <Rect id="rect1" width="200" height="200">
          <fill>
             <SolidColor color="#FFFFCC"/>
          </fill>
          <stroke>
             <SolidColorStroke color="#660099" weight="2"/>
          </stroke>
     </Rect>
</Graphic>
```

When using FXG documents as components, you specify the tag to be the name of the FXG file, just as you would do with an ActionScript or MXML component.

The following application uses the GraphicComp.fxg file as a component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/GraphicCompMain.mxml -->
<s:Application backgroundColor="0xFFFFFF"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <comps:GraphicComp id="graphic1"/>
</s:Application>
```

FXG documents encapsulate nearly all of their functionality in a single document. FXG documents cannot reference other FXG documents or MXML documents. However, they can reference the following external resources:

- External bitmap files. For an example that embeds an external image, see “Bitmap graphics in FXG and MXML graphics” on page 1772.
- Platform fonts by family name and style name.

Like MXML graphics, FXG tags have an implicit depth order. The order in which elements are defined defines their depth. Each tag is effectively drawn above its previous sibling. Children are drawn on top of their parents.

Adobe Creative Suite tools can be used to convert SVG to FXG. To do this, open the SVG file in the tool and export it as an FXG file.

FXG syntax

The root of an FXG document file is a `<Graphic>` tag. An FXG document can include zero or more containers (such as Group) and graphic elements, as well as a single library that can include any number of definitions.

The `<Graphic>` tag can only appear once in an FXG document. While the document cannot contain other `<Graphic>` elements, it can contain other elements such as `<Rect>`, `<Ellipse>`, `<Path>`, and `<BitmapImage>`. The `<Graphic>` tag can also optionally contain a single child `<Library>` tag and/or a single `<mask>` tag. These elements must appear before any other tag. If both are present, then the `<Library>` tag must come first.

You cannot specify an ID for the root `<Graphic>` tag in an FXG document.

FXG documents use the following language namespace:

```
http://ns.adobe.com/fxg/2008
```

When you create an FXG document, you must also add the `version` attribute to the `<Graphic>` root tag. The currently supported version is 2. For example:

```
:Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
```

To use an FXG document in an application, you treat is as any other component. In your application, you define a namespace that matches the location of the FXG document. This namespace definition includes a prefix. For example:

```
<s:Application
    ...
    xmlns:comps="comps.*">
```

You then use that prefix to reference the FXG document as a component; for example:

```
<comps:MyGraphicComp id="graphic1"/>
```

FXG data types

The following table describes the basic data types that you can use in an FXG document:

Data Type	Description
angle	An arbitrary number specified as degrees. Angles are clockwise.
color	A numerical RGB specification in hexadecimal notation. You specify a color by using a pound sign (#) immediately followed by six hexadecimal characters. You cannot use the 0x notation that is supported in MXML graphics.
coordinate	Represents a length in the local coordinate system that is the given distance from the origin of the local coordinate system along the relevant axis (the x-axis for x coordinates, the y-axis for y coordinates).
identifier	A text string that matches the regular expression [A-Za-z][A-Za-z0-9_]*.
integer	An optional sign character (+ or -) followed by one or more digits 0 through 9. If the sign character is not present, the number is non-negative. Unless stated otherwise for a particular attribute or property, the range for an integer ranges from -2147483648 to 2147483647.
length	A distance measurement. The format of a length is a number.
number	Specified either in decimal notation, or in scientific notation. Decimal notation consists of either an integer or an optional sign character, followed by zero or more digits, followed by a dot (.), followed by one or more digits. Scientific notation consists of a decimal-number immediately followed by the letter "e" or "E" immediately followed by an integer. Unless stated otherwise for a particular attribute or property, a number has the capacity for at least a single-precision floating point number and has a range of -3.4e+38F to +3.4e+38F.
percentage	A number immediately followed by a percentage sign (%). Percentage values are always relative to another value; for example, a length. Attributes or properties that allow percentage values also define the reference distance measurement to which the percentage refers.

Tags not supported by FXG

While FXG is syntactically very similar to MXML, FXG does not define equivalents for all the MXML language tags. It does, however, include the `<Definition>`, `<Library>` and `<Private>` elements, which are equivalent to the MXML language tags of the same names. FXG does not define any other MXML language tags. The MXML language tags not defined by FXG include the following tags:

- `<fx:Binding>`
- `<fx:Component>`
- `<fx:Declaration>`
- `<fx:Metadata>`
- `<fx:Model>`
- `<fx:Reparent>`
- `<fx:Repeater>`
- `<fx:Script>`
- `<fx:State>`
- `<fx:states>`
- `<fx:Style>`

FXG syntax also does not include any ActionScript 3.0 built-in primitive tags. The list of built-in tags that are not defined by FXG includes the following:

- `<fx:Array>`
- `<fx:Boolean>`

- `<fx:Class>`
- `<fx:Date>`
- `<fx:Function>`
- `<fx:int>`
- `<fx:Number>`
- `<fx:Object>`
- `<fx:String>`
- `<fx:uint>`
- `<fx:XML>`
- `<fx:XMLList>`

FXG does not support data binding. If you try to use the data binding short-hand syntax in an FXG document, the compiler treats it as a literal String value.

Using FXG in applications built with Flex

When you use an FXG document as a component in your application, the compiler optimizes the document into a subclass of the spark.core.SpriteVisualElement class. Specifically, the compiler maps the FXG elements to SWF graphics primitive tags and links only a light-weight, Sprite-based class into your application.

The SpriteVisualElement class extends the Sprite class. It adds support for sizing, positioning, and alpha when used in an application built with Flex.

To use FXG as a component in your applications:

1 Store the FXG document in a location where the compiler can find it. This can be in the same directory as the application, or in a separate location. If you store the FXG document in a separate location, you must add the location to the compiler's source path.

2 Declare a namespace for the component. For example, if the FXG file is in the same directory as the main application that uses it, you can declare a namespace of "*". If the FXG file is in a directory called comps, you can declare a namespace of "comps.*".

3 Add a tag in your application that declares the component inside a Spark container. (You cannot use a MX container as a direct parent of an FXG component.) You can set DisplayObject properties on the tag, such as `x` and `y`, `alpha`, `height` and `width`. You can also add event handlers that are supported by the SpriteVisualElement class on the tag.

The following example creates multiple instances of the star.fxg component, and sets properties on each of those instances:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/OptimizedFXGExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="*"
    width="500" height="300">
    <s:Group>
        <comps:star height="100" width="100" x="50" y="50"/>
        <comps:star rotationX="20" height="70" width="70" x="150" y="50" alpha=".75"/>
        <comps:star rotationX="40" height="50" width="50" x="220" y="50" alpha=".5"/>
        <comps:star rotationX="60" height="30" width="30" x="270" y="50" alpha=".3"/>
        <comps:star rotationX="80" height="10" width="10" x="300" y="50" alpha=".1"/>
    </s:Group>
</s:Application>
```

The following is the FXG component that is used by the previous example:

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- fxg/star.fxg -->
<fxg:Graphic xmlns:fxg="http://ns.adobe.com/fxg/2008" version="2">
    <fxg:Path x="9.399" y="10.049" data="M 82.016 78.257 L 51.895 69.533 L 27.617 89.351 L 
26.621 58.058 L 0.231 41.132 L 29.749 30.52 L 37.714 0.241 L 56.944 24.978 L 88.261 23.181 L 
70.631 49.083 Z">
        <fxg:fill>
            <fxg:SolidColor color="#FFFFFF"/>
        </fxg:fill>
        <fxg:stroke>
            <fxg:SolidColorStroke 
                caps="none" 
                color="#4769C4" 
                joints="miter" 
                miterLimit="4" 
                weight="20"/>
        </fxg:stroke>
    </fxg:Path>
</fxg:Graphic>
```

FXG documents use only the *.fxg filename suffix. You cannot have another file of the same name with an *.mxml or *.as suffix in the same directory.

You can use ActionScript to instantiate an FXG component. When you do this, you declare the FXG tag to be of type SpriteVisualElement. You then add the component to the display list by calling the `addElement()` method, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/OptimizedFXGActionScriptExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="500" height="300"
    creationComplete="drawStar()">
    <fx:Script>
        <![CDATA[
        import spark.core.SpriteVisualElement;
        private var myStar:SpriteVisualElement;
        private function drawStar():void {
            // Create new instances of star.fxg as if it were a local component.
            for (var i:int = 0; i<4; i++) {
                myStar = new star();
                myStar.x = 50 + (i*75);
                myStar.y = 50;
                myStar.height = 100 - (i*30);
                myStar.width = 100 - (i*30);
                myStar.alpha = 1 - (i*.2);
                myStar.rotationX = 20 + (i*20);
                addElement(myStar);
            }
        }
        ]]>
    </fx:Script>
</s:Application>
```

Generating FXG

When you export a graphic file from a tool such as Adobe Illustrator in the FXG format, the tool writes a *.fxg file with a `<Graphic>` root tag.

If you are given the option to select a version of FXG, select version 2.

You can ignore the information in the `<Private>` code block, as long as you also leave the its namespace declaration. The contents of the `<Private>` block must be well-formed and valid XML but is not rendered.

You can optionally remove the `<Private>` tag and its contents from the FXG document. Some tools export FXG with additional syntax so that the file can be re-edited. An FXG document exported from Illustrator, for example, includes a `<Private>` block of code that is used by Illustrator if you want to edit the file again. If you do not expect to edit the file in Illustrator again, then you can remove the `<Private>` block.

Converting FXG elements to MXML graphics

In general, you should use an FXG document as a standalone component in your applications. This gives you the greatest amount of memory optimization, and lets you reuse your FXG files in other parts of the same application, or in other applications. In addition, by keeping the FXG document separate from the application, you can edit and export the FXG again from a graphics tool.

One disadvantage to using an FXG file as a standalone component is that you cannot get a reference to individual elements in it. The FXG component itself is a “black box”; you cannot interact with individual elements from your application or other components. You can, however, still apply effects, resize, position, and trigger events from interaction with the FXG component itself.

In some cases, you might want to add the FXG output directly into your application. You can do this, but you must convert the FXG syntax to MXML graphics.

After converting FXG elements to MXML graphics, you can get a reference to the various graphic elements, just as you can with any other MXML tags. This is useful if you want to apply special effects or events to what were previously FXG elements. After you convert the FXG file to MXML graphics, you cannot re-edit the FXG file in a graphics tool. It is now part of MXML.

To convert an FXG document to MXML graphics:

1 Start by copying and pasting the FXG code into your MXML document.

2 Update the namespaces:

   1 Change the FXG namespace (http://ns.adobe.com/fxg/2008) to the Spark component namespace (library://ns.adobe.com/flex/spark). There can be only one language namespace per document.

   2 If the FXG fragment uses additional tool specific namespaces, you can add these to the root of the MXML file. For example, FXG exported from Adobe Illustrator typically includes the following namespace declarations:

   ```
   xmlns:ai="http://ns.adobe.com/ai/2008"
   xmlns:d="http://ns.adobe.com/fxg/2008/dt"
   ```

   These namespaces do not impact the rendering of the document, but must be retained to keep the XML document valid as tool-specific private attributes will be prefixed in these namespaces.

3 Change or remove any namespace prefixes on the `<Library>`, `<Definition>`, and `<Private>` elements. They must use the MXML 2009 language namespace (http://ns.adobe.com/mxml/2009). If you remove a namespace prefix on the tag, you can usually just add the "fx:" prefix to each of these elements.

4 Move the `<Library>` tag and any `<Definition>` elements it contains to the top of the MXML document. If there is already a `<fx:Library>` tag in the MXML document, move just the `<Definition>` elements into it and remove the FXG `<Library>` tag.

5 For library definitions, add the "fx:" prefix to their tags in the body of the MXML file. In MXML 2009 documents, library definitions are actually in the language namespace. For example, the `<fx:Definition name="BlueCircle">` tag references to a definition that must also be in the "fx" namespace; for example, `<fx:BlueCircle>`.

6 Move the `<Private>` tag to the end of the MXML document or remove it. You do not need to include the `<Private>` tag at all, but it can contain useful information about the tool that generated the FXG.

7 Update the namespace prefix of the FXG elements. There is likely to be an "s:" prefix already mapped to the Spark component namespace on the MXML document's root tag (xmlns:s="library://ns.adobe.com/flex/spark"). Add the "s:" prefix to all of the FXG graphical elements. For example, change `<Graphic>` to `<s:Graphic>`.

8 For FXG 1.0 documents only:

   a Convert `<TextGraphic>` elements to `<s:RichText>`. This does not apply to FXG 2.0 documents, which use the `<RichText>` tag. In that case, you just add the "s:" prefix to the `<RichText>` tag to make it conform to MXML syntax.

   b Convert `<BitmapGraphic>` elements to `<s:BitmapImage>`. This does not apply to FXG 2.0 documents, which use the `<BitmapImage>` tag. In that case, you just add the "s:" prefix to the `<BitmapImage>` tag to make it conform to MXML syntax.

The following file, exported in FXG format from Adobe Illustrator, includes the private information as well as the `<Graphic>` root tag. The contents of this *.fxg file are being shown so that you can then see in a subsequent example how the contents are converted to MXML graphics.

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- assets/skins/SimpleBox.fxg -->
<Graphic version="1.0"
    viewHeight="30"
    viewWidth="100"
    ai:appVersion="14.0.0.367"
    d:id="1"
    xmlns="http://ns.adobe.com/fxg/2008"
    xmlns:ai="http://ns.adobe.com/ai/2008"
    xmlns:d="http://ns.adobe.com/fxg/2008/dt">
  <Library/>
  <Group
    x="-0.296875"
    y="-0.5" d:id="2"
    d:type="layer"
    d:userLabel="Layer 1">
    <Group d:id="3">
      <Rect x="0.5" y="0.5" width="100" height="30" ai:knockout="0">
        <fill>
          <LinearGradient x="0.5" y="15.5" scaleX="100" rotation="-0">
            <GradientEntry color="#ffffff" ratio="0"/>
            <GradientEntry ratio="1"/>
          </LinearGradient>
        </fill>
        <stroke>
          <SolidColorStroke
            color="#0000ff"
            caps="none"
            weight="1"
            joints="miter"
            miterLimit="4"/>
        </stroke>
      </Rect>
    </Group>
  </Group>
  <Private>
    <ai:PrivateElement d:ref="#1">
      <ai:SaveOptions>
        <ai:Dictionary>
          <ai:DictEntry name="preserveGradientPolicy" value="3" valueType="Integer"/>
          <ai:DictEntry name="rasterizeResolution" value="72" valueType="Integer"/>
          <ai:DictEntry name="clipToActiveArtboard" value="1" valueType="Boolean"/>
          <ai:DictEntry name="downsampleLinkedImages" value="0" valueType="Boolean"/>
          <ai:DictEntry name="preserveFilterPolicy" value="3" valueType="Integer"/>
          <ai:DictEntry name="preserveTextPolicy" value="3" valueType="Integer"/>
          <ai:DictEntry name="writeImages" value="1" valueType="Boolean"/>
          <ai:DictEntry name="includeXMP" value="0" valueType="Boolean"/>
          <ai:DictEntry name="aiEditCap" value="1" valueType="Boolean"/>
          <ai:DictEntry name="versionKey" value="1" valueType="Integer"/>
          <ai:DictEntry name="includeSymbol" value="0" valueType="Boolean"/>
        </ai:Dictionary>
      </ai:SaveOptions>
      <ai:DocData base="SimpleBox.assets/images"/>
      <ai:Artboards
        originOffsetH="0"
        originOffsetV="30"
        rulerCanvasDiffH="50.5"
        rulerCanvasDiffV="-14.5"
        zoom="17.17">
        <ai:Artboard active="1" index="0" right="100" top="30"/>
        <ai:ArtboardsParam all="0" range="" type="0"/>
      </ai:Artboards>
    </ai:PrivateElement>
    <ai:PrivateElement d:ref="#2">
      <ai:LayerOptions colorType="ThreeColor">
        <ai:ThreeColor blue="257" green="128.502" red="79.31"/>
      </ai:LayerOptions>
    </ai:PrivateElement>
    <ai:PrivateElement
        ai:hashcode="769d7bac08ad6bdcf80f40fca11df6c0"
        d:ref="#3">
      <ai:Rect height="30" knockout="0" width="100" x="0.5" y="0.5">
        <ai:Stroke colorType="ThreeColor" miterLimit="4" weight="1">
          <ai:ThreeColor blue="1"/>
        </ai:Stroke>
        <ai:Fill colorType="Gradient">
          <ai:Gradient
            gradientType="linear"
            length="100" originX="0.5"
            originY="15.5">
            <ai:GradientStops>
              <ai:GradientStop colorType="GrayColor" rampPoint="0">
                <ai:GrayColor/>
              </ai:GradientStop>
              <ai:GradientStop colorType="GrayColor" rampPoint="100">
                <ai:GrayColor gray="1"/>
              </ai:GradientStop>
            </ai:GradientStops>
          </ai:Gradient>
        </ai:Fill>
        <ai:ArtStyle/>
      </ai:Rect>
    </ai:PrivateElement>
  </Private>
</Graphic>
```

To convert this example to MXML graphics:

- Remove the `<Library/>` tag. This example does not contain any `<Definition>` elements.
- Remove the `<Private>` block.
- Move the two Illustrator-specific namespaces to the application's root tag:

  ```
  xmlns:ai="http://ns.adobe.com/ai/2008"
  xmlns:d="http://ns.adobe.com/fxg/2008/dt"
  ```

- Remove the FXG-specific namespace:

  ```
  "http://ns.adobe.com/fxg/2008"
  ```

- Add the "s:" prefix to all elements.
- Because the "d" and "ai" namespace definitions are intact, you do not have to remove the attributes specific to these namespaces. The attributes `type` and `userLabel`, for example, appear on the `<Group>` tag from the original FXG file. To further simplify this example, though, you could remove them because Flex ignores anything in those namespaces.

For example, this:

```
<s:Group x="-0.296875" y="-0.5" d:id="2" d:type="layer" d:userLabel="Layer 1">
```

Becomes this:

```
<s:Group x="-0.296875" y="-0.5">
```

The following example application contains the converted FXG from the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/ResultsMXMLGraphicsApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:ai="http://ns.adobe.com/ai/2008"
    xmlns:d="http://ns.adobe.com/fxg/2008/dt">

    <s:Graphic version="1.0"
        viewHeight="30"
        viewWidth="100"
        ai:appVersion="14.0.0.367"
        d:id="1">
      <s:Group x="-0.296875" y="-0.5" d:id="2" d:type="layer" d:userLabel="Layer 1">
        <s:Group d:id="3">
          <s:Rect x="0.5" y="0.5" width="100" height="30" ai:knockout="0">
            <s:fill>
              <s:LinearGradient x="0.5" y="15.5" scaleX="100" rotation="-0">
                <s:GradientEntry color="#ffffff" ratio="0"/>
                <s:GradientEntry ratio="1"/>
              </s:LinearGradient>
            </s:fill>
            <s:stroke>
              <s:SolidColorStroke
                color="#0000ff"
                caps="none"
                weight="1"
                joints="miter"
                miterLimit="4"/>
            </s:stroke>
          </s:Rect>
        </s:Group>
      </s:Group>
    </s:Graphic>
</s:Application>
```

Coordinate systems

FXG defines two coordinate system concepts: the document coordinate system, and the user coordinate system.

The *document coordinate system* refers to the coordinate system of the root tag. By default, its origin sits at the top left of the document, and extends downward along the positive y axis, and to the right along the positive x axis. 1 unit corresponds to 1 pixel on the screen.

The *user coordinate system* refers to the coordinate system defined on any individual tag in the document. In FXG, the user coordinate system at the root `<Graphic>` tag is identical to the document coordinate system.

By default, each grouping instance element and graphic element defines its user coordinate system to be identical to that of its parent. Any geometry transform defined on the tag (through attributes or child transform elements) transforms its parent's user coordinate system into a new system.

All attributes of elements are defined in units of the current user coordinate system. As a result, the coordinates of the segments of a path are relative to its coordinate system. To determine the position of the path segments in document coordinates, you would multiply its x and y by the geometry transform of the path and each of its parent elements until you reached the root graphic tag.

Some fills and strokes have their own user coordinate system. As with Groups, the default coordinate system is aligned with the coordinate system of their most immediate parent instance. As appropriate, fills and strokes support geometry transforms that can modify their coordinate space.

Sizing FXG components

FXG elements use their `height` and `width` properties to determine the size of the graphic. The graphic tag scales to the values of the `height` and `width` properties. The `viewWidth` and `viewHeight` properties define the space that the graphic takes up. When you set these values, the content is not scaled.

If you specify a `viewWidth` and `viewHeight` that is larger than the natural size of the content, the graphic takes up more space than its visual size. The result is a boundary around the graphic.

You can also specify a `viewWidth` and `viewHeight` that is smaller than the natural size of the content. You might do this if your graphic has chrome such as a border that extends past the edges of the graphic. In this case, be sure to turn off clipping in your layout.

Optimizing FXG

Use the techniques described in this section when working with FXG.

Composite path versus rounded rectangle

Composite paths create a cleaner look because there are no extra pixels inside the corners. Rounded rectangles, on the other hand, have extra pixels due to anti-aliasing. Rounded rectangles convert to a `<Rect>` element in FXG, which is preferable because a `<Rect>` element is easier to manipulate than a compex path.

Move alpha values from the element to the fill or stroke

When exporting FXG files from a graphics editor, the `alpha` property is sometimes placed on the graphic element tag. If you convert the FXG to MXML, this element uses its own DisplayObject, which can be computationally expensive. To avoid this, move any `alpha` properties from the element tag down to the stroke or fill. If both the stroke/fill and the element have `alpha` values, multiply the stroke and fill `alpha` values by the element's `alpha` value. For example, if the stroke and element `alpha` values are both .5, then remove the element's `alpha` property and set the stroke's `alpha` to .25 (.5 x .5).

Round alpha values to 2 decimal places

Exported FXG files sometimes use percentage values for `alpha` values. The visual difference between an `alpha` value with two decimal places and a rounded `alpha` value is negligible. To make your code more readable, round off `alpha` values to 2 decimal places. For example, round an `alpha` value of 0.05882352941176471 to 0.06.

Remove blendMode="normal"

By default on graphic elements, the `blendMode` is "auto". When the `blendMode` property is "auto", Flash Player correctly determines whether the element needs to use the "layer" `blendMode` based on the `alpha` value.

Identifying elements of an FXG file

Sometimes it can be hard to look at FXG tags and determine what that element looks like. An easy way to do this is to copy your FXG into a `<s:Group>` tag in an MXML application. Then, change the x values of each element so that they no longer overlap. For example, if your skin is 45 pixels wide and is comprised of three `<Rect>` elements, increase the `x` value of the second `<Rect>` by 50 and the third `<Rect>` by 100. When you compile and run the application, you can see each layer spread separately.

An alternative is to toggle the `visible` property of different elements so that you can isolate a particular element.

Placing shapes with odd stroke weights (mobile applications only)

For a given shape, strokes straddle the shape edges as opposed to being on the inside or outside of an edge. For example, take a vertical line starting from 0,0 to 0,100 with a 1-pixel wide stroke. That stroke stretches in the x-axis from -.5 to .5. However, because you cannot draw an element at a fractional pixel, Flash Player draws an anti-aliased line 2 pixels wide to approximate the fractional position.

This is true only on mobile applications. For desktop applications, Flash Player pixel-snaps stroke segments.

To counteract this anti-aliasing, place your shapes with odd numbered stroke weights on a half pixel boundary. For example, place a `<Rect>` with a `<SolidColorStroke>` of weight 1 at 10,10 at 10.5,10.5 instead.

An alternative is to leave all of your stroked shapes at integer pixels and shift the entire FXG component by .5 in the x and y positions. If possible, use filled `<Rect>` elements in place of stroked `<Rect>` or `<Line>` elements. If you have a `<Rect>` element with a fill and a stroke, try to split it into two filled `<Rect>` elements This technique works only if the fill is completely opaque. If you have a `<Line>` element, try using a filled `<Rect>` element instead. For example, if you have a horizontal `<Line>` element, create a filled `<Rect>` element with a height equal to the `<Line>` element's stroke weight.

Reduce anti-aliasing due to stage quality

Mobile applications built in Flex are currently limited to using `StageQuality.MEDIUM`. When designing skins using FXG and/or programmatic graphics, watch for anti-aliasing artifacts in rounded corners. The mobile skins mitigate this issue by using FXG Path data with fills instead strokes to draw curves.

To create the appearance of a rounded rectangle at a 1px stroke in Fireworks:

1 Draw a filled rounded rectangle.

2 Copy the rectangle, position it 1px down and to the left of the original rectangle and reduce its width and height by 2px.

3 Select both rectangles, right click and choose Combine Path > Punch.

Use 45 degree angles

Try to keep all angles at 45 degrees. This helps to reduce the visual artifacts from anti-aliasing.

MXML graphics

MXML graphics are a collection of classes that define graphic elements in an application. MXML graphics tags support interactivity. They can be children of any container or group, can be declared as children of the root Application tag, and can be added to the application's display list.

Most MXML graphics classes are in the mx.graphics and spark.primitives packages. These classes include shapes, strokes, colors, fills, and gradients.

MXML graphics tags have an implicit depth order. The order in which elements are defined defines their depth. Each tag is effectively drawn above its previous sibling. Children are drawn above their parents.

The following example uses MXML graphics to draw a filled rectangle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/GraphicCompMainMXML.mxml -->
<s:Application backgroundColor="0xFFFFFF"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Graphic>
         <s:Rect id="rect1" width="200" height="200">
              <s:fill>
                 <s:SolidColor color="0xFFFFCC"/>
              </s:fill>
              <s:stroke>
                 <s:SolidColorStroke color="0x660099" weight="2"/>
              </s:stroke>
         </s:Rect>
    </s:Graphic>
</s:Application>
```

Because the classes that make up MXML graphics are part of the Flex SDK, you can reference them from within ActionScript or from other components. The following example changes the rectangle's color when you click the button:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/ReferenceGraphics.mxml -->
<s:Application backgroundColor="0xFFFFFF"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Graphic>
         <s:Rect id="rect1" width="200" height="200">
              <s:fill>
                 <s:SolidColor id="sc1" color="0xFFFFCC"/>
              </s:fill>
              <s:stroke>
                 <s:SolidColorStroke color="0x660099" weight="2"/>
              </s:stroke>
         </s:Rect>
    </s:Graphic>

    <s:Button label="Change Color" click="sc1.color=0xCCFFFF"/>
</s:Application>
```

All MXML graphics classes have backing ActionScript implementations. This means that every MXML tag has a corresponding ActionScript class that the compiler maps the tag to during compilation. The table in "Graphics classes and elements" on page 1715 lists the tags and their backing ActionScript classes. You can use the classes in ActionScript directly in your `<fx:Script>` blocks to draw vector based graphics instead of the MXML tags. For more information, see Using the drawing API.

Paths in FXG and MXML graphics

Paths define a filled graphic tag that draws a series of path segments. In vector graphics, a path is a series of points connected by straight or curved line segments. Together the lines form an image.

In FXG, you use a `<Path>` tag to define a vector shape constructed from a set of line segments. The MXML graphics equivalent of the FXG `<Path>` tag is the Path class. In MXML, you use the `<s:Path>` tag.

There are several different types of path segments you can use: line, cubic bezier, and quadratic bezier. Typically, the first tag of a path definition moves the pen to the starting position of the graphic. You then use the segments to draw the lines of the graphic.

To define a series of segments in a path, you use the path's `data` property. You pass this property a series of instructions using shorthand syntax. Each value is preceded by a letter that acts as a segment identifier. The segment identifier indicates which type of segment to draw with the numeric values that follow it.

The following table describes the shorthand syntax used by the Path tag's `data` property:

Segment Type	**Segment Identifier**	**Parameters**	**Example**
Close path	`Z/z`	n/a	`Z` Closes off the path.
Cubic Bezier	`C/c`	`c1X c1Y c2X c2Y x y`	`C 45 50 20 30 10 20` Curve to (10, 20), with the first control point at (45, 50) and the second control point at (20, 30).
Cubic Bezier (without control points)	`S/s`	`c2X c2Y x y`	S 20 30 10 20 Curve to (10, 20), with the second control point at (20, 30). The first control point is assumed to be the reflection of the second control point on the previous command relative to the current point.
Horizontal line	`H/h`	`x`	`H 100` Horizontal line to 100.
Line	`L/l`	`x y`	`L 50 30` Line to (50, 30).
Move	`M/m`	`x y`	`M 10 20` Move pen position to (10, 20).
Quadratic Bezier	`Q/q`	`cX cY x y`	`Q 110 45 90 30` Curve to (90, 30) with the control point at (110, 45).
Quadratic Bezier (without control points)	T/t	x y	T 90 30 Curve to (90, 30). The control point is the reflection of the control point on the previous command relative to the current point.
Vertical line	`V/v`	`y`	`V 100` Vertical line to 100.

You can use an uppercase or lowercase letter for the segment identifiers. If you specify a segment identifier using the uppercase letter, then the positioning is absolute. If you specify a segment identifier using the lowercase letter, then the positioning is relative.

When using segments, you do not need to specify the starting position of the pen; the x and y coordinate of the starting point is defined by the current pen position.

After drawing a line segment, the current pen position becomes the x and y coordinates of the end point of the line. You can use the Move segment type to reposition the pen without drawing a line.

The Path syntax is nearly identical to the SVG path syntax. This makes it easy to convert SVG paths to FXG or MXML graphics. There is one exception: FXG and MXML graphics do not support an “A” or Arc segment. To create an arc (elliptical, parabolic, or otherwise), use the the cubic Bezier and quadratic Bezier control points to achieve any sort of arc.

Paths can contain effects such as transforms, filters, blend modes, and masks.

FXG examples

The following example uses the `<Path>` tag to draw a rectangle in FXG:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimplePathShorthandExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <s:Panel title="Rectangle Example" height="75%" width="75%">
           <comps:SimplePathComp x="10" y="10"/>
     </s:Panel>
</s:Application>
```

The following is the FXG component that is used by the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/SimplePathComp.fxg -->
<fx:Graphic xmlns:fx="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Use shorthand syntax to draw the rectangle. -->
    <fx:Path data="M 0 0 L 0 100 L 100 100 L 100 0 L 0 0">
         <!-- Define the border color of the rectangle. -->
         <fx:stroke>
              <fx:SolidColorStroke color="#888888"/>
         </fx:stroke>
    </fx:Path>
</fx:Graphic>
```

The following example draws two arrows in FXG, one in relative coordinates and one in absolute coordinates.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/ArrowExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Arrow Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
          <s:HGroup>
              <!-- Use the ArrowAbsolute FXG component which uses absolute coordinates. -->
               <comps:ArrowAbsolute/>

              <!-- Use the ArrowRelative FXG component which uses relative coordinates. -->
               <comps:ArrowRelative/>
          </s:HGroup>
     </mx:Panel>
</s:Application>
```

The following file defines an arrow graphic as an FXG graphic using absolute coordinates:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/ArrowAbsolute.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Use Use compact syntax with absolute coordinates. -->
    <Path data="
         M 20 0
         C 50 0 50 35 20 35
         L 15 35
         L 15 45
         L 0 32
         L 15 19
         L 15 29
         L 20 29
         C 44 29 44 6 20 6">
         <!-- Define the border color of the arrow. -->
         <stroke>
              <SolidColorStroke color="#888888"/>
         </stroke>
         <!-- Define the fill for the arrow. -->
         <fill>
              <LinearGradient rotation="90">
                   <GradientEntry color="#000000" alpha="0.8"/>
                   <GradientEntry color="#FFFFFF" alpha="0.8"/>
              </LinearGradient>
         </fill>
    </Path>
</Graphic>
```

The following file defines an arrow graphic as an FXG graphic using relative coordinates:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/ArrowRelative.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Use compact syntax with relative coordinates. -->
    <Path data="
              m 20 0
              c 30 0 30 35 0 35
              l -5 0
              l 0 10
              l -15 -13
              l 15 -13
              l 0 10
              l 5 0
              c 24 0 24 -23 0 -23">
         <!-- Define the border color of the arrow. -->
         <stroke>
              <SolidColorStroke color="#888888"/>
         </stroke>
         <!-- Define the fill for the arrow. -->
         <fill>
              <LinearGradient rotation="90">
                   <GradientEntry color="#000000" alpha="0.8"/>
                   <GradientEntry color="#FFFFFF" alpha="0.8"/>
              </LinearGradient>
         </fill>
    </Path>
</Graphic>
```

MXML graphics examples

The following MXML graphics example uses the `<s:Path>` tag to draw a rectangle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimplePathShorthandExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:Panel title="Rectangle MXML Graphics Example" height="75%" width="75%">
        <s:Path x="10" y="10" data="M 0 0 L 0 100 L 100 100 L 100 0 L 0 0">
             <!-- Define the border color of the rectangle. -->
             <s:stroke>
                  <s:SolidColorStroke color="0x888888"/>
             </s:stroke>
        </s:Path>
     </s:Panel>
</s:Application>
```

The following MXML graphics example defines two arrows, one with relative and the other with absolute positioning:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/ArrowExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Arrow MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
          <s:HGroup>
              <!-- Use absolute coordinates. -->
                <s:Graphic>
                    <!-- Use Use compact syntax with absolute coordinates. -->
                    <s:Path data="
                         M 20 0
                         C 50 0 50 35 20 35
                         L 15 35
                         L 15 45
                         L 0 32
                         L 15 19
                         L 15 29
                         L 20 29
                         C 44 29 44 6 20 6">
                         <!-- Define the border color of the arrow. -->
                         <s:stroke>
                              <s:SolidColorStroke color="0x888888"/>
                         </s:stroke>
                         <!-- Define the fill for the arrow. -->
                         <s:fill>
                              <s:LinearGradient rotation="90">
                                   <s:GradientEntry color="0x000000" alpha="0.8"/>
                                   <s:GradientEntry color="0xFFFFFF" alpha="0.8"/>
                              </s:LinearGradient>
                         </s:fill>
                    </s:Path>
                </s:Graphic>
              <!-- Use relative coordinates. -->
                <s:Graphic>
                    <!-- Use compact syntax with relative coordinates. -->
                    <s:Path data="
                              m 20 0
                              c 30 0 30 35 0 35
                              l -5 0
                              l 0 10
                              l -15 -13
```

```
                                        l 15 -13
                                        l 0 10
                                        l 5 0
                                        c 24 0 24 -23 0 -23">
                                   <!-- Define the border color of the arrow. -->
                                   <s:stroke>
                                        <s:SolidColorStroke color="0x888888"/>
                                   </s:stroke>
                                   <!-- Define the fill for the arrow. -->
                                   <s:fill>
                                        <s:LinearGradient rotation="90">
                                             <s:GradientEntry color="0x000000" alpha="0.8"/>
                                             <s:GradientEntry color="0xFFFFFF" alpha="0.8"/>
                                        </s:LinearGradient>
                                   </s:fill>
                              </s:Path>
                         </s:Graphic>
                   </s:HGroup>
     </mx:Panel>
</s:Application>
```

Libraries and symbols in FXG and MXML graphics

A symbol is a named grouping tag that can be made up of other symbols and graphic elements. You can define a symbol once and reuse it any number of times in an FXG document by using the `<Library>` and `<Definition>` elements. In MXML graphics, you use the MXML file's `<fx:Library>` and `<fx:Definition>` tags.

In FXG, the `<Library>` tag contains `<Definition>` elements. The `<Library>` tag can only be placed as a child of the root tag of an FXG document, just as the `<fx:Library>` tag can only be a child of the root tag in an MXML file. Symbols defined in a library can be referenced by name anywhere in the document.

In FXG, the `<Library>` tag must be the first child of the root tag.

The FXG `<Definition>` tag and the MXML `<fx:Definition>` tag can contain only a Group or other symbols. You cannot nest definitions, as the following example shows:

```
<!-- This is illegal. -->
<Library>
    <Definition name="A">
        <Definition name="B"/>
    </Definition>
</Library>
```

You can, however, reference symbols inside definitions. Symbols can be used anywhere a Group can be used, as the following example shows:

```
<!-- This is OK. -->
<Library>
    <Definition name="A/>
    <Definition name="B">
        <A/>
    </Definition>
</Library>
```

All attributes that are legal on an instance group are also legal on a symbol.

Libraries can have any number of definitions. Each definition defines a named symbol that you can reuse in that document. For example, you can define a rectangle shape in your library, and then use that rectangle shape any number of times in the document. The named symbol can optionally contain effects such as color transforms, filters, blend modes, or masks. These elements can appear in any order in FXG.

For more information about using the `<fx:Library>` and `<fx:Definition>` language tags in MXML, see *ActionScript 3.0 Reference for the Adobe Flash Platform*.

FXG examples

The following example uses three FXG components. Each component includes a library definition of an object, and uses that tag multiple times to make up its shape.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LibraryExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Library Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
          <s:VGroup>
            <comps:YellowShape/>
            <comps:BlueShape/>
            <comps:RedShape/>
          </s:VGroup>
     </mx:Panel>
</s:Application>
```

The following file defines the YellowShape FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/YellowShape.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
     <Library>
          <Definition name="YellowRect">
               <Group>
                   <Rect height="10" width="10">
                         <stroke>
                              <SolidColorStroke color="#000000" weight=".5"/>
                         </stroke>
                         <fill>
                              <SolidColor color="#FFFF00"/>
                         </fill>
                   </Rect>
                </Group>
          </Definition>
     </Library>
     <YellowRect x="0" y="0"/>
     <YellowRect x="0" y="12"/>
     <YellowRect x="12" y="0"/>
     <YellowRect x="12" y="12"/>
</Graphic>
```

The following file defines the BlueShape FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/BlueShape.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
     <Library>
          <Definition name="BlueRect">
               <Group>
                   <Rect height="10" width="10">
                         <stroke>
                              <SolidColorStroke color="#000000" weight=".5"/>
                         </stroke>
                         <fill>
                              <SolidColor color="#66CCFF"/>
                         </fill>
                   </Rect>
                </Group>
          </Definition>
     </Library>
     <BlueRect x="0" y="0"/>
     <BlueRect x="12" y="0"/>
     <BlueRect x="24" y="0"/>
     <BlueRect x="36" y="0"/>
</Graphic>
```

The following file defines the RedShape FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/RedShape.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
     <Library>
          <Definition name="RedRect">
               <Group>
                   <Rect height="10" width="10">
                         <stroke>
                              <SolidColorStroke color="#000000" weight=".5"/>
                         </stroke>
                         <fill>
                              <SolidColor color="#FF3333"/>
                         </fill>
                   </Rect>
                </Group>
          </Definition>
     </Library>
     <RedRect x="0" y="0"/>
     <RedRect x="12" y="0"/>
     <RedRect x="12" y="12"/>
     <RedRect x="24" y="12"/>
</Graphic>
```

MXML graphics examples

The following example draws three groups of shapes using MXML graphics that are defined in the library:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LibraryExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <fx:Library>
          <fx:Definition name="YellowRect">
               <s:Group>
                   <s:Rect height="10" width="10">
                         <s:stroke>
                              <s:SolidColorStroke color="#000000" weight=".5"/>
                         </s:stroke>
                         <s:fill>
                              <s:SolidColor color="#FFFF00"/>
                         </s:fill>
                   </s:Rect>
                </s:Group>
          </fx:Definition>
          <fx:Definition name="BlueRect">
               <s:Group>
                   <s:Rect height="10" width="10">
                         <s:stroke>
                              <s:SolidColorStroke color="#000000" weight=".5"/>
                         </s:stroke>
                         <s:fill>
                              <s:SolidColor color="#66CCFF"/>
                         </s:fill>
                   </s:Rect>
                </s:Group>
          </fx:Definition>
          <fx:Definition name="RedRect">
               <s:Group>
                   <s:Rect height="10" width="10">
                         <s:stroke>
                              <s:SolidColorStroke color="#000000" weight=".5"/>
                         </s:stroke>
                         <s:fill>
                              <s:SolidColor color="#FF3333"/>
                         </s:fill>
                   </s:Rect>
                </s:Group>
          </fx:Definition>
     </fx:Library>
     <mx:Panel title="Library MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
          <s:VGroup>
            <s:Graphic>
             <fx:YellowRect x="0" y="0"/>
             <fx:YellowRect x="0" y="12"/>
             <fx:YellowRect x="12" y="0"/>
             <fx:YellowRect x="12" y="12"/>
            </s:Graphic>
            <s:Graphic>
             <fx:BlueRect x="0" y="0"/>
             <fx:BlueRect x="12" y="0"/>
             <fx:BlueRect x="24" y="0"/>
             <fx:BlueRect x="36" y="0"/>
            </s:Graphic>

            <s:Graphic>
             <fx:RedRect x="0" y="0"/>
             <fx:RedRect x="12" y="0"/>
             <fx:RedRect x="12" y="12"/>
             <fx:RedRect x="24" y="12"/>
            </s:Graphic>

     </s:VGroup>
     </mx:Panel>
</s:Application>
```

Groups in FXG and MXML graphics

Groups contain content items. In MXML, these items are typically of type IGraphicElement (the base interface class for graphic elements), but can also include effects such as the `<transform>`, `<filters>`, `<mask>` elements, and other `<Group>` elements. In MXML graphics, items inside a group can be any MXML tags.

You define a group in an FXG document by using the `<Group>` tag. In MXML graphics, you use the `<s:Group>` tag, which corresponds to the Group class. This class implements the IGraphicElementContainer interface.

The order of graphical objects inside a group determines their depth order when rendered. The last object defined in a group is the top-most tag of the depth order.

A group can be inside a graphic or another group. When a group is inside a graphic or other group, it is considered an instance group. Instance groups can optionally contain features such as transforms, filters, and masks. The order that these children appear does not matter. Graphical effects such as these are rendered based on a predefined order, as described in "Effects in FXG and MXML graphics" on page 1776.

A Group can also be used inside a definition in the library section. When a group is inside a definition, it is considered a symbol definition. Groups inside libraries (symbol definition groups) cannot contain effects such as masks or transforms.

A group defines a new local coordinate space for its immediate child elements.

For more information about using the `<s:Group>` tag in MXML, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

FXG example

The following FXG example uses the YellowShape FXG component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/GroupCoordinateExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

     <mx:Panel title="Group Coordinate Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
       <s:Group>
            <comps:YellowShape/>
       </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the YellowShape FXG component used in the previous example. The component is defined in a `<Group>` tag inside a `<Library>`, which means it is a symbol definition group:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/YellowShape.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
     <Library>
          <Definition name="YellowRect">
               <Group>
                   <Rect height="10" width="10">
                         <stroke>
                              <SolidColorStroke color="#000000" weight=".5"/>
                         </stroke>
                         <fill>
                              <SolidColor color="#FFFF00"/>
                         </fill>
                   </Rect>
                </Group>
          </Definition>
     </Library>
     <YellowRect x="0" y="0"/>
     <YellowRect x="0" y="12"/>
     <YellowRect x="12" y="0"/>
     <YellowRect x="12" y="12"/>
</Graphic>
```

MXML graphics example

The following MXML graphics example uses the `<s:Group>` tag to define a group of yellow shapes:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/GroupCoordinateExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <fx:Library>
          <fx:Definition name="YellowRect">
               <s:Group>
                   <s:Rect height="10" width="10">
                         <s:stroke>
                              <s:SolidColorStroke color="#000000" weight=".5"/>
                         </s:stroke>
                         <s:fill>
                              <s:SolidColor color="#FFFF00"/>
                         </s:fill>
                   </s:Rect>
                </s:Group>
          </fx:Definition>
     </fx:Library>
     <mx:Panel title="Group Coordinate MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
       <s:Group>
                 <fx:YellowRect x="0" y="0"/>
                 <fx:YellowRect x="0" y="12"/>
                 <fx:YellowRect x="12" y="0"/>
                 <fx:YellowRect x="12" y="12"/>
       </s:Group>
     </mx:Panel>
</s:Application>
```

Shapes in FXG and MXML graphics

FXG and MXML graphics define the following basic shapes:

- Rectangles
- Ellipses
- Lines

The MXML graphics classes for the basic shapes are defined in the spark.primitives package.

The default border for any shape is to have a stroke with a line width of 0. Therefore, you must define a stroke when you draw a shape, otherwise the shape will be invisible when rendered. You can optionally fill the shape with a fill.

Rectangles

In FXG, the `<Rect>` tag draws a rectangular shape; in MXML graphics, use the `<s:Rect>` tag or the spark.primitives.Rect class. You define the height and width of a rectangle. You can also define the stroke, or border, of the rectangle, and a fill.

When drawing rectangles, you have a great deal of control over the curvature of the corners of the rectangle. The FXG `<Rect>` tag and the `<s:Rect>` MXML tag class include properties such as `topLeftRadiusX`, `topLeftRadiusY`, `bottomRightRadiusX`, and `bottomRightRadiusY`. Convenience properties, `radiusX` and `radiusY`, let you set the radii of all corners.

FXG examples

The following FXG example draws a simple rectangle with a 1 point black stroke:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleRectExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Rectangle Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:SimpleRectComp />
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the SimpleRectComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/SimpleRectComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
   <Rect height="100" width="200">
         <stroke>
              <SolidColorStroke color="#000000" weight="1"/>
         </stroke>
   </Rect>
</Graphic>
```

The following FXG-based example draws two rectangles with strokes and fills. One rectangle has rounded corners:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/RectExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Rectangle Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:VGroup>
            <comps:FilledRectComp/>
            <comps:RoundRectComp/>
        </s:VGroup>
     </mx:Panel>
</s:Application>
```

The following file defines the FilledRectComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/FilledRectComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Draw rectangle with square corners. -->
   <Rect height="100" width="200">
         <stroke>
              <SolidColorStroke color="#000000" weight="2"/>
         </stroke>
         <fill>
              <RadialGradient>
                        <GradientEntry color="#0056FF" ratio="0" alpha=".5"/>
                        <GradientEntry color="#00CC99" ratio=".33" alpha=".5"/>
                        <GradientEntry color="#ECEC21" ratio=".66" alpha=".5"/>
              </RadialGradient>
         </fill>
   </Rect>
</Graphic>
```

The following file defines the RoundRectComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/RoundRectComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Draw rectangle with rounded corners. -->
    <Rect height="100" width="200" radiusX="25" radiusY="25">
         <stroke>
              <SolidColorStroke color="#000000" weight="2"/>
         </stroke>
         <fill>
              <RadialGradient>
                        <GradientEntry color="#0056FF" ratio="0" alpha=".5"/>
                        <GradientEntry color="#00CC99" ratio=".33" alpha=".5"/>
                        <GradientEntry color="#ECEC21" ratio=".66" alpha=".5"/>
              </RadialGradient>
         </fill>
    </Rect>
</Graphic>
```

MXML graphics examples

The following MXML graphics example draws a simple rectangle with a 1 point black stroke:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleRectExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <mx:Panel title="Rectangle MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
           <s:Rect height="100" width="200">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="1"/>
                 </s:stroke>
           </s:Rect>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following MXML graphics example draws two rectangles with strokes and fills. One rectangle has rounded corners:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/RectExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

     <mx:Panel title="Rectangle MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:VGroup>
           <s:Rect height="100" width="200">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="2"/>
                 </s:stroke>
                 <s:fill>
                      <s:RadialGradient>
                                <s:GradientEntry color="0x0056FF" ratio="0" alpha=".5"/>
                                <s:GradientEntry color="0x00CC99" ratio=".33" alpha=".5"/>
                                <s:GradientEntry color="0xECEC21" ratio=".66" alpha=".5"/>
                      </s:RadialGradient>
                 </s:fill>
           </s:Rect>
          <s:Rect height="100" width="200" radiusX="25" radiusY="25">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="2"/>
                 </s:stroke>
                 <s:fill>
                      <s:RadialGradient>
                                <s:GradientEntry color="0x0056FF" ratio="0" alpha=".5"/>
                                <s:GradientEntry color="0x00CC99" ratio=".33" alpha=".5"/>
                                <s:GradientEntry color="0xECEC21" ratio=".66" alpha=".5"/>
                      </s:RadialGradient>
                 </s:fill>
            </s:Rect>
        </s:VGroup>
     </mx:Panel>
</s:Application>
```

For more information, see the Rect class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Ellipses

In FXG, use the `<Ellipse>` tag to draw circles and ellipses; in MXML graphics, use the `<s:Ellipse>` tag or the spark.primitives.Ellipse class.

To define the dimensions of the ellipse, use the `height` and `width` attributes. These attributes are relative to the graphic's upper left corner. If you set the `height` and `width` attributes to the same value, the ellipse is a circle.

FXG examples

The following FXG-based example draws a simple circle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleEllipseExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Ellipse Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:CircleComp />
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the CircleComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/CircleComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Ellipse height="200" width="200">
          <stroke>
               <SolidColorStroke color="#000000" weight="1"/>
          </stroke>
    </Ellipse>
</Graphic>
```

The following FXG-based example draws an ellipse with a fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/EllipseExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Ellipse Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:FilledCircleComp/>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the FilledCircleComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/FilledCircleComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Ellipse height="100" width="250">
          <stroke>
               <SolidColorStroke color="#000000" weight="2"/>
          </stroke>
        <fill>
           <RadialGradient>
             <GradientEntry color="#0056FF" ratio="0" alpha=".5"/>
             <GradientEntry color="#00CC99" ratio=".33" alpha=".5"/>
             <GradientEntry color="#ECEC21" ratio=".66" alpha=".5"/>
         </RadialGradient>
        </fill>
    </Ellipse>
</Graphic>
```

MXML graphics examples

The following MXML graphics example draws a simple circle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleEllipseExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <mx:Panel title="Ellipse MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <s:Ellipse height="200" width="200">
                  <s:stroke>
                       <s:SolidColorStroke color="0x000000" weight="1"/>
                  </s:stroke>
            </s:Ellipse>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following MXML graphics example draws an ellipse with a fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/EllipseExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Ellipse MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <s:Ellipse height="100" width="250">
                  <s:stroke>
                       <s:SolidColorStroke color="0x000000" weight="2"/>
                  </s:stroke>
                <s:fill>
                   <s:RadialGradient>
                     <s:GradientEntry color="0x0056FF" ratio="0" alpha=".5"/>
                     <s:GradientEntry color="0x00CC99" ratio=".33" alpha=".5"/>
                     <s:GradientEntry color="0xECEC21" ratio=".66" alpha=".5"/>
                 </s:RadialGradient>
                </s:fill>
            </s:Ellipse>
        </s:Group>
     </mx:Panel>
</s:Application>
```

For more information, see the Ellipse class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Lines

To draw a line from one point to another in FXG, use the `<Line>` tag; in MXML graphics, use the `<s:Line>` tag or the spark.primitives.Line class.

The default width of a line is 0; you must define a stroke so that the line is visible. Strokes can be one of the following:

- SolidColorStroke
- LinearGradientStroke
- RadialGradientStroke

You can optionally define the color, joint types, and other properties of the line.

FXG examples

The following FXG-based example draws a simple 1 point black line with the SolidColorStroke child:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleLineExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Line Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:LineComp/>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the LineComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/LineComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
   <Line xFrom="0" xTo="100" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="1"/>
         </stroke>
   </Line>
</Graphic>
```

The following FXG-based example draws a series of lines with varying widths:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LineExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Line Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:MultipleLineComp x="20" y="20"/>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the MultipleLineComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/MultipleLineComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Line xFrom="0" xTo="0" yFrom="0" yTo="100">
         <!-- Define the border color of the line. -->
         <stroke>
              <SolidColorStroke color="#000000" weight="1" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="6" xTo="6" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="1" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="12" xTo="12" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="2" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="20" xTo="20" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="3" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="30" xTo="30" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="5" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="43" xTo="43" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="8" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="58" xTo="58" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="13" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="84" xTo="84" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="21" caps="square"/>
         </stroke>
    </Line>
    <Line xFrom="123" xTo="123" yFrom="0" yTo="100">
         <stroke>
              <SolidColorStroke color="#000000" weight="34" caps="square"/>
         </stroke>
    </Line>
</Graphic>
```

To use a LinearGradientStroke or RadialGradientStroke in FXG, you add one or more `<GradientEntry>` elements as children. These gradients are used to define the fill of the stroke. The following example defines a line whose color shifts from blue to yellow:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LinearGradientLineExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Linear Gradient Line Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:LinearGradientLineComp x="20" y="20"/>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the LinearGradientLineComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/LinearGradientLineComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Line xFrom="0" xTo="200" yFrom="0" yTo="0">
         <stroke>
              <LinearGradientStroke weight="20" caps="square">
                    <GradientEntry color="#0056FF" ratio="0" alpha=".5"/>
                    <GradientEntry color="#ECEC21" ratio=".66" alpha=".5"/>
              </LinearGradientStroke>
         </stroke>
    </Line>
</Graphic>
```

MXML graphics examples

The following MXML graphics example draws a simple 1 point black line with the SolidColorStroke child:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleLineExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <mx:Panel title="Line MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
           <s:Line xFrom="0" xTo="100" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="1"/>
                 </s:stroke>
           </s:Line>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following MXML graphics example draws a series of lines with varying widths:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LineExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Line MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="30" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group x="20" y="20">
            <s:Line xFrom="0" xTo="0" yFrom="0" yTo="100">
                 <!-- Define the border color of the line. -->
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="1" caps="square"/>
                 </s:stroke>
            </s:Line>
            <s:Line xFrom="6" xTo="6" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="1" caps="square"/>
                 </s:stroke>
            </s:Line>
            <s:Line xFrom="12" xTo="12" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="2" caps="square"/>
                 </s:stroke>
            </s:Line>
            <s:Line xFrom="20" xTo="20" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="3" caps="square"/>
                 </s:stroke>
            </s:Line>
            <s:Line xFrom="30" xTo="30" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="5" caps="square"/>
                 </s:stroke>
            </s:Line>
            <s:Line xFrom="43" xTo="43" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="8" caps="square"/>
```

```
                </s:stroke>
            </s:Line>
            <s:Line xFrom="58" xTo="58" yFrom="0" yTo="100">
                <s:stroke>
                    <s:SolidColorStroke color="0x000000" weight="13" caps="square"/>
                </s:stroke>
            </s:Line>
            <s:Line xFrom="84" xTo="84" yFrom="0" yTo="100">
                <s:stroke>
                    <s:SolidColorStroke color="0x000000" weight="21" caps="square"/>
                </s:stroke>
            </s:Line>
            <s:Line xFrom="123" xTo="123" yFrom="0" yTo="100">
                <s:stroke>
                    <s:SolidColorStroke color="0x000000" weight="34" caps="square"/>
                </s:stroke>
            </s:Line>
        </s:Group>
     </mx:Panel>
</s:Application>
```

To use a LinearGradientStroke or RadialGradientStroke in MXML graphics, you add one or more `<s:GradientEntry>` tags as children. These gradients are used to define the fill of the stroke. The following MXML graphics example defines a line whose color shifts from blue to yellow:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LinearGradientLineExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Linear Gradient Line MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
            <s:Group x="20" y="20">
                <s:Line xFrom="0" xTo="200" yFrom="0" yTo="0">
                    <s:stroke>
                        <s:LinearGradientStroke weight="20" caps="square">
                              <s:GradientEntry color="0x0056FF" ratio="0" alpha=".5"/>
                              <s:GradientEntry color="0xECEC21" ratio=".66" alpha=".5"/>
                        </s:LinearGradientStroke>
                    </s:stroke>
                </s:Line>
            </s:Group>
     </mx:Panel>
</s:Application>
```

For more information, see the Line class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Text in FXG and MXML graphics

Text is rendered as a graphic element similar to paths and shapes, but with a restricted subset of rendering options. Text elements are always rendered using a solid fill color, modified by any opacity, blend mode, and color transformation defined by parent elements, and clipped to any clipping content defined on its parent elements. Text content is only filled, not stroked.

To use text in an FXG 2.0 document, use the `<RichText>` tag. In FXG 1.0, use the `<TextGraphic>` tag.

In MXML, you can use the RichText, RichEditableText, or Label classes to render text. For more information on using text controls in MXML, see "MX text controls" on page 805.

In FXG, the `<RichText>` tag supports the following child elements:

- `<content>`
- `<transform>`

To specify text in a `<RichText>` tag, you use a single `<content>` child tag. The following FXG-based example shows several methods of using text in a `<RichText>` tag:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/FXGRichTextExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*" height="100%" width="100%">

     <mx:Panel title="RichText Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:HGroup>
            <s:Group>
                <comps:TextComp/>
            </s:Group>
            <s:Group>
                <comps:JustifiedTextComp/>
            </s:Group>
            <s:Group>
                <comps:FormattedTextComp/>
            </s:Group>
        </s:HGroup>
     </mx:Panel>
</s:Application>
```

The following file defines the TextComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/TextComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <RichText>
       <content>Hello World!</content>
    </RichText>
</Graphic>
```

The following file defines the JustifiedTextComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/JustifiedTextComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <RichText textAlign="justify" width="100">
       <content>This is a block of text that is justified, so you can see formatting
applied.</content>
    </RichText>
</Graphic>
```

The following file defines the FormattedTextComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/FormattedTextComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <RichText>
       <content>This text is <span fontWeight="bold">BOLD</span>,<br/>
       this text is <span textDecoration="underline">UNDERLINED</span>,<br/>
       and this text is <span fontStyle="italic">ITALIC</span>.
       <p>This is a new paragraph.</p></content>
    </RichText>
</Graphic>
```

The `<content>` child tag supports the following child elements:

- `<p>`
- `<span>`
- `<br>`
- `<tcy>`
- `<a>`
- `<img>`
- `<tab>`
- `<linkNormalFormat>`
- `<linkHoverFormat>`
- `<linkActiveFormat>`

The `<content>` child tag can also take raw text.

If you do not explicitly use a `<p>` tag as the first tag in a `<content>` tag, then it is implied.

If you apply style attributes to the `<RichText>` tag (such as `fontFamily="arial"`), then the `<span>` tag is implied on the content of the `<RichText>` tag. The following example has multiple text formats, with one implied span and one explicit span:

```
<RichText fontFamily="Verdana" fontWeight="bold">
    <content>Hello, <span fontWeight="normal">World</span></content>
</RichText>
```

The contents of the `<RichText>` tag support a set of style attributes that define formatting. These styles are a subset of the MXML RichText class. Some styles are applied at the character and paragraph level, and some are applied at the container level. Container-level attributes (whose level is listed as including `<RichText>`), affect the entire text flow when applied to the container. Paragraph-level attributes (whose level is listed as including `<p>`), affect the paragraph when applied to a `<p>` tag. And character-level attributes (whose level is listed as including `<span>`) affect only the current span when applied to a `<span>` tag.

The following table describes the formatting attributes supported by FXG:

Attribute	**Description**	**Level**
alignmentBaseline	Specifies which of the baselines of the line containing the element the dominantBaseline snaps to, thus determining the vertical position of the element in the line. The default value is useDominantBaseline. Valid values are roman, ascent, descent, ideographicTop, ideographicCenter, ideographicBottom, and useDominantBaseline.	<RichText>, <p>, or <span>
backgroundAlpha	Alpha (transparency) value for the background. A value of 0 is fully transparent, and a value of 1 is fully opaque. The default value is 1.	<RichText>, <p>, or <span>
backgroundColor	Background color of the text. Can be either transparent, or an integer containing three 8-bit RGB components. The default value is transparent.	<RichText>, <p>, or <span>
baselineShift	Indicates the baseline shift for the element in pixels. The element is shifted perpendicular to the baseline by this amount. In horizontal text, a positive baseline shift moves the element up and a negative baseline shift moves the element down. The default value is 0.0, indicating no shift. A value of superscript shifts the text up by an amount specified in the font, and applies a transform to the fontSize also based on preferences in the font. A value of subscript shifts the text down by an amount specified in the font, and also transforms the fontSize. A percent value shifts the text by a percentage of the fontSize attribute. The minimum/maximum percent are -1000%/1000%. A Number shifts the text by that number of pixels. The minimum/maximum Number values are -1000/1000.	<RichText>, <p>, or <span>
blockProgression	Controls the direction in which lines are stacked. In Latin text, this is tb, because lines start at the top and proceed downward. In vertical Chinese or Japanese, this is rl, because lines should start at the right side of the container and proceed leftward. Valid values are tb and rl.	<RichText> or <p>
breakOpportunity	Controls where a line can legally break. Valid values are auto, any, none, or all. A value of auto means line breaking opportunities are based on standard Unicode character properties, such as breaking between words and on hyphens. A value of any indicates that the line may end at any character. This value is typically used when Roman text is embedded in Asian text and it is desirable for breaks to happen in the middle of words. A value of none means that no characters in the range are treated as line break opportunities. A value of all means that all characters in the range are treated as mandatory line break opportunities; this results in one character per line. The default value is auto.	<RichText>, <p>, or <span>
color	Sets the color of the text. The default value is 0x000000.	<RichText>, <p>, or <span>

Attribute	Description	Level
columnCount	Defines the number of columns in the RichText element. The column number overrides the other column settings. The value can be an Integer, or auto if unspecified. If it's an integer, the range of legal values is 0 to 50. If columnCount is not specified, but columnWidth is, then columnWidth is used to create as many columns as can fit in the container. The default value is auto.	<RichText>
columnGap	Defines th space between columns in pixels. Does not include space before the first column or after the last column. Use the padding properties to define that. Legal values range from 0 to 1000. The default value is 0.	<RichText>
columnWidth	Defines the width of columns in pixels. If you specify the width of the columns, but not the count, Flex creates as many columns of that width as possible given the the container width and columnGap settings. Any remainder space is left after the last column. Legal values are 0 to 8000. The default value is auto.	<RichText>
digitCase	Defines the case for digits. Valid values are default, lining, and oldStyle. A value of default uses the normal digit case from the font. A value of lining uses the lining digit case from the font. A value of oldStyle uses the old style digit case from the font. The default value is default.	<RichText>, <p>, or <span>
digitWidth	Specifies how wide digits will be when the text is set. Valid values are proportional, tabular, and default. A value of proportional means that the proportional widths from the font are used, and different digits will have different widths. A value of tabular means that every digits has the same width. A value of default means that the normal width from the font is used. The default value is default.	<RichText>, <p>, or <span>
direction	Controls the dominant writing direction for the paragraphs (left-to-right or right-to-left), and how characters with no implicit writing direction, such as punctuation, are treated. Valid values are ltr and rtl. The default value is ltr.	<RichText> or <p>

Attribute	Description	Level
`dominantBaseline`	Specifies which of the baselines of the element snaps to the alignmentBaseline to determine the vertical position of the element on the line. Valid values are `auto`, `roman`, `ascent`, `descent`, `ideographicTop`, `ideographicCenter`, and `ideographicBottom`. A value of `auto` gets resolved based on the `textRotation` of the span and the locale of the parent paragraph. A `textRotation` of `rotate270` resolves to `ideographicCenter`. A locale of Japanese ("ja") or Chinese ("zh-XX", "zh_XX", etc), resolves to `ideographicCenter`, whereas all others are resolved to `roman`. The default value is `auto`.	`<RichText>`, `<p>`, or `<span>`
`firstBaselineOffset`	Specifies the position of the first line of text in the container (first in each column) relative to the top of the container. The first line may appear at the position of the line's ascent, or below by the `lineHeight` of the first line. Or it may be offset by a pixel amount. Valid values are `auto`, `ascent`, `lineHeight`, and a Number. The default value (auto) specifies that the line top be aligned to the container top inset. The baseline that this property refers to is deduced from the container's locale as follows: `ideographicBottom` for Chinese and Japanese locales, `roman` otherwise. Minumum/maximum values are 0/1000. The default value is `auto`.	`<RichText>`
`fontFamily`	Defines the font family name used to render the text. The font depends on the glyphs that are being rendered and the fonts that are available on the client system. For most platforms, the default value is Arial.	`<RichText>`, `<p>`, or `<span>`
`fontSize`	Sets the size of the glyphs that are used to render the text, specified in point sizes. The default value is 12. The minimum value is 1. The maximum value is 500.	`<RichText>`, `<p>`, or `<span>`
`fontStyle`	Italicizes text. Valid values are `normal` and `italic`. The default value is `normal`.	`<RichText>`, `<p>`, or `<span>`
`fontWeight`	Bolds text. Valid values are `normal` and `bold`. The default value is `normal`.	`<RichText>`, `<p>`, or `<span>`
`justificationRule`	Defines the justifier. A value of `eastAsian` enables justification for Japanese. The default value is `auto`. A value of `auto` is resolved based on the locale of the paragraph. Values for Japanese ("ja") and Chinese ("zh-XX", "zh_XX", etc) resolve to `eastAsian`, while all other locales resolve to space.	`<RichText>` or `<p>`

Attribute	Description	Level
justificationStyle	Defines the justification style. A value of auto is resolved based on the locale of the paragraph. Currently, all locales resolve to pushInKinsoku, however, this value is only used in conjunction with a justificationRule value of eastAsian, so is only applicable to "ja" and all "zh" locales. A value of prioritizeLeastAdjustment bases justification on either expanding or compressing the line, whichever gives a result closest to the desired width. A value of pushInKinsoku bases justification on compressing kinsoku at the end of the line, or expanding it if there is no kinsoku or if that space is insufficient. A value of pushOutOnly bases justification on expanding the line. Valid values are auto, prioritizeLeastAdjustment, pushInKinsoku, and pushOutOnly. The default value is auto.	<RichText> or <p>
kerning	Applies pair kerning to text. Valid values are on, off, and auto. If on, pair kerns are honored. If off, there is no font-based kerning applied. If auto, kerning is applied to all characters except Kanji, Hiragana or Katakana. The default value is auto.	<RichText>, <p>, or <span>
leadingModel	Specifies the leading basis (baseline to which the lineHeight property refers) and the leading direction (which determines whether the lineHeight property refers to the distance of a line's baseline from that of the line before it or the line after it). The default value is auto which is resolved based on locale. Locale values of Japanese ("ja") and Chinese ("zh-XX", "zh_XX", etc) resolve auto to ideographicTopDown and other locales resolve to romanUp. Valid values are auto, romanUp, ideographicTopUp, ideographicCenterUp, ascentDescentUp, ideographicTopDown, and ideographicCenterDown.	<RichText> or <p>
ligatureLevel	Controls which ligatures in the font will be used. Valid values are minimum, common, uncommon, and exotic. A value of minimum turns on rlig, common is rlig + clig + liga; uncommon is rlig + clig + liga + dlig; exotic is rlig + clig + liga + dlig + hlig. You cannot turn the various ligature features on independently. The default value is common.	<RichText>, <p>, or <span>
lineBreak	Determines whether the text wraps the lines at the edge of the enclosing <RichText> element. Set to toFit to wrap the lines. Set to explicit to break the lines only at a Unicode line end character (such as a newline or line separator). Valid values are toFit and explicit. The default value is toFit.	<RichText>

Attribute	Description	Level
`lineHeight`	Defines the leading, or the distance from the previous line's baseline to the current line, in points. This can be specified in absolute pixels, or as a percentage of the point size. The default value is 120%. The minimum value for percent or number is 0.	`<RichText>`, `<p>`, or `<span>`
`lineThrough`	Whether to apply a strike through to the text. Set to `true` to apply a line through the text; otherwise set to `false`. The default value is `false`.	`<RichText>`, `<p>`, or `<span>`
`locale`	Defines the locale of the text. This controls case transformations and shaping. Standard locale identifiers as described in Unicode Technical Standard #35 are used. For example en, en_US and en-US are all English, ja is Japanese. Locale applied at the paragraph and higher level impacts resolution of "auto" values for `dominantBaseline`, `justificationRule`, `justificationStyle` and `leadingModel`. See individual attributes for resolution values.	`<RichText>`, `<p>`, or `<span>`
`marginBottom`	Defines the space after the paragraph. As in CSS, adjacent vertical space collapses. No margin is necessary if the paragraph falls at the bottom of the `<RichText>` tag. The default value is 0. The minimum value is 0.	`<RichText>` or `<p>`
`marginLeft`	Defines the indentation applied to the left edge of the `<RichText>` tag, measured in pixels. The default value is 0.	`<RichText>` or `<p>`
`marginRight`	Defines the indentation applied to the right edge of the `<RichText>` tag, measured in pixels. The default value is 0.	`<RichText>` or `<p>`
`marginTop`	Defines the space before the paragraph. As in CSS, adjacent vertical space collapses. Given two adjoining paragraphs (A, B), where A has `marginBottom` 12 and B has `marginBottom` 24, the total space between the paragraphs is 24, the maximum of the two, and not 36, the sum. If the paragraph is at the top of the column, no extra space is left for the margin. The default value is 0. The minimum value is 0.	`<RichText>` or `<p>`
`paddingBottom`	Defines the inset from bottom edge to content area, in pixels. The default value is 0. The minumum/maximum values are 0/1000.	`<RichText>`
`paddingLeft`	Defines the inset from left edge to content area, in pixels. The default value is 0. The minumum/maximum values are 0/1000.	`<RichText>`
`paddingRight`	Defines the inset from right edge to content area, in pixels. The default value is 0. The minumum/maximum values are 0/1000.	`<RichText>`
`paddingTop`	Defines the inset from top edge to content area, in pixels. The default value is 0. The minumum/maximum values are 0/1000.	`<RichText>`

Attribute	**Description**	**Level**
`paragraphEndIndent`	Defines the indentation applied to the end edge of a paragraph (right edge if direction is ltr, left edge otherwise). Measured in pixels. Legal values range from 0 to 1000. Default is 0.	`<RichText>` or `<p>`
`paragraphSpaceAfter`	Defines the amount of whitespace that appears the paragraph. As in CSS, adjacent vertical space collapses (see note for paragraphSpaceBefore). No "space after" is necessary if the paragraph falls at the bottom of the RichText. Legal values range from 0 to 1000. The default value is 0. The minimum value is 0.	`<RichText>` or `<p>`
`paragraphSpaceBefore`	Specifies the amount of whitespace that appears above a paragraph. As in CSS, adjacent vertical space collapses. For two adjoining paragraphs (A, B), where A has paragraphSpaceAfter 12 and B has paragraphSpaceBefore 24, the total space between the paragraphs will be 24, the maximum of the two, and not 36, the sum. If the paragraph comes at the top of the column, no extra space is left before it. Legal values range from 0 to 1000. The default value is 0. The minimum value is 0.	`<RichText>` or `<p>`
`paragraphStartIndent`	Defines the indentation applied to the start edge of the paragraph (left edge if direction is ltr, right edge otherwise). Measured in pixels. Legal values range from 0 to 1000. The default value is 0.	`<RichText>` or `<p>`
`tabStops`	Defines an array of tab stops. For more information, see the FXG 2.0 Specification.	`<RichText>` or `<p>`
`textAlpha`	Sets the opacity level of the text, ranging from 0 (completely transparent) to 1 (completely opaque). The default value is 1.	`<RichText>`, `<p>`, or `<span>`
`textAlign`	Aligns text relative to the text box edges. Valid values are `left`, `right`, `center`, and `justify`. The default value is `left`.	`<RichText>` or `<p>`
`textAlignLast`	Aligns the last line of the paragraph. To make a paragraph set all lines justified, set the `textAlign` and `textAlignLast` attributes to `justify`. Valid values are `left`, `center`, `right`, and `justify`. The default value is `left`.	`<RichText>` or `<p>`
`textDecoration`	Underlines text. Valid values are `none` and `underline`. The default value is `none`.	`<RichText>`, `<p>`, or `<span>`
`textIndent`	Indent the first line of text in a paragraph. The indent is relative to the left margin. Measured in pixels. The default value is 0. This attribute can be negative.	`<RichText>` or `<p>`

Attribute	Description	Level
`textJustify`	Applies when `justificationRule` is space. A value of `interWord` spreads justification space out to spaces in the line. A value of `distribute` spreads it out to letters as well as spaces. The default value is `interWord`. Valid values are `interWord` and `distribute`.	`<RichText>` or `<p>`
`textRotation`	Controls the rotation of the text, in ninety degree increments. Valid values are `auto`, `rotate0`, `rotate90`, `rotate180`, and `rotate270`. The default value is `auto`.	`<RichText>`, `<p>`, or `<span>`
`trackingLeft`	Adds space to the left of each character. Can be a percent value or a Number. A Number tracks by a pixel amount, with the minimum/maximum values -100/1000. A percent value is the percentage of the `fontSize` attribute. Legal values for percentages are -100% to 1000%. Negative values bring characters closer together. The default value is 0.	`<RichText>`, `<p>`, or `<span>`
`trackingRight`	Adds space to the right of each character. Can be a percent value or a Number. A Number tracks by a pixel amount, with the minimum/maximum values -100/1000. A percent value is the percentage of the `fontSize` attribute. Legal values for percentages are -100% to 1000%. Negative values bring characters closer together. The default value is 0.	`<RichText>`, `<p>`, or `<span>`

Attribute	Description	Level
typographicCase	Controls the case in which the text appears. Valid values are default, capsToSmallCaps, uppercase, lowercase, and lowercaseToSmallCaps. A value of default does not apply any case changes.A value of smallCaps converts all characters to uppercase and applies c2sc. A value of uppercase and lowercase are case conversions.A value of caps turns on case.A value of lowercaseToSmallCaps converts all characters to uppercase, and for those characters which have been converted, applies c2sc. The default value is default.	<RichText>, <p>, or <span>
verticalAlign	Sets the vertical alignment of the lines within the container. Valid values are top, middle, bottom, justify, and inherit. The lines might appear at the top of the container, centered within the container, at the bottom, or evenly spread out across the depth of the container. The default value is top.	<RichText>
whiteSpaceCollapse	Determines how whitespace such as line feeds, newlines, and tabs are treated. Valid values are collapse and preserve. Set to collapse to convert line feeds, newlines, and tabs to spaces, collapse adjacent spaces to one, and trim leading and trailing whitespace. Set to preserve to maintain whitespace without changes.	<RichText>, <p>, or <span>

Not all of the properties in MXML text-based controls are supported by the <RichText> tag in FXG, and not all properties of FXG are supported by the MXML text-based controls. For example, the RichText class in MXML does not have the margin-related properties that the <RichText> tag in FXG has.

The <RichText> tag is not optimized when an FXG document is compiled. It is the only FXG element that is not optimized because there are no low-level Flash Player instructions for RichText.

If you do not specify a the width or height properties of the <RichText> tag, or if the specified width or height is 0, the width and height are calculated based on the text content. This is done using the following logic for horizontal text:

- If the width is specified, but the height is not, the container height is set to the height required to fit the text. Text wraps to the width of the container, and the total height of the text becomes the container height.
- If the width is not specified, then the text breaks only at line breaks and at the paragraph end. The width of the container is then set to the width of the longest line.
- If neither width nor height are specified, the width is set as described above, and then the height is set based on the width.
- If the height is specified, and the text exceeds what will fit, then the remaining text is preserved but does not appear in the container; it is clipped. You can choose to add scroll bars to view the additional text, but scroll bars are not part of FXG.

Fills in FXG and MXML graphics

Fills and strokes generally define a default bounding area that they fill, clipped by the path they are filling. Fills and strokes can define their own user coordinate space, which defaults to the coordinate space of the object they are filling. A fill or stroke can modify its coordinate space with transform attributes and child elements, similar to a Group, that is concatenated with its parent's transforms to define a fill region in document coordinates.

To define a fill in FXG, you use the `<fill>` tag as the child of a graphic element, such as a Rect. In MXML graphics, you use the `<s:fill>` tag, or a class that implements the mx.graphics.IFill interface. Fill elements and tags can contain a single tag that defines the type of fill.

The following types of fills are supported by FXG and MXML graphics:

- SolidColor
- RadialGradient
- LinearGradient
- BitmapFill

SolidColor fills

A solid color fill fills a path or shape with a single solid color or solid color with opacity. The value of the `color` property is a hexadecimal number, such as #FF00FF. In FXG, you cannot use the 0x notation that is supported in MXML graphics.

In FXG, you use the `<SolidColor>` tag to create a solid color fill. In MXML graphics, you use the `<s:SolidColor>` tag.

FXG example

The following FXG-based example creates a rectangle with a SolidColor fill at 50% opacity:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SolidColorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Solid Color Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
           <s:Group>
                <comps:SolidColorComp/>
           </s:Group>
     </mx:Panel>
</s:Application>
```

The following FXG file defines the SolidColorComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/SolidColorComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Draw rectangle that is filled with a SolidColor. -->
   <Rect height="100" width="200">
         <stroke>
              <SolidColorStroke color="#000000" weight="2"/>
         </stroke>
         <fill>
              <SolidColor color="#00FF00" alpha=".5"/>
         </fill>
   </Rect>
</Graphic>
```

MXML graphics example

The following example draws a rectangle and fills it with a SolidColor fill at 50% opacity:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SolidColorExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <mx:Panel title="Solid Color MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
           <s:Group>
                <!-- Draw rectangle that is filled with a SolidColor. -->
               <s:Rect height="100" width="200">
                     <s:stroke>
                          <s:SolidColorStroke color="0x000000" weight="2"/>
                     </s:stroke>
                     <s:fill>
                          <s:SolidColor color="0x00FF00" alpha=".5"/>
                     </s:fill>
               </s:Rect>
           </s:Group>
     </mx:Panel>
</s:Application>
```

For more information, see the SolidColor class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

LinearGradient fills

A linear gradient fills a path or shape with a continuously smooth color transition between a list of colors along a vector. With gradient fills, you can use the `spreadMethod` property to define a repetition of the patterns.

In FXG, you use the `<LinearGradient>` tag to define a linear gradient. In MXML graphics, you use the `<s:LinearGradient>` tag, or the mx.graphics.LinearGradient class.

FXG example

The following FXG-based example uses a LinearGradient fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LinearGradientExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Linear Gradient Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
           <s:Group>
                <comps:LinearGradientComp/>
            </s:Group>
     </mx:Panel>
</s:Application>
```

The following FXG file defines the LinearGradientComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/LinearGradientComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Draw rectangle that is filled with a LinearGradient. -->
   <Rect height="100" width="200">
         <stroke>
              <SolidColorStroke color="#000000" weight="2"/>
         </stroke>
         <fill>
              <LinearGradient>
                <GradientEntry color="#0056FF" ratio="0" alpha=".5"/>
                <GradientEntry color="#00CC99" ratio=".33" alpha=".5"/>
                <GradientEntry color="#ECEC21" ratio=".66" alpha=".5"/>
              </LinearGradient>
         </fill>
   </Rect>
</Graphic>
```

MXML graphics example

The following MXML graphics example draws a rectangle and fills it with a LinearGradient fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/LinearGradientExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Linear Gradient MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
           <s:Group>
                <!-- Draw rectangle that is filled with a LinearGradient. -->
               <s:Rect height="100" width="200">
                     <s:stroke>
                          <s:SolidColorStroke color="0x000000" weight="2"/>
                     </s:stroke>
                     <s:fill>
                          <s:LinearGradient>
                            <s:GradientEntry color="0x0056FF" ratio="0" alpha=".5"/>
                            <s:GradientEntry color="0x00CC99" ratio=".33" alpha=".5"/>
                            <s:GradientEntry color="0xECEC21" ratio=".66" alpha=".5"/>
                          </s:LinearGradient>
                     </s:fill>
               </s:Rect>
            </s:Group>
     </mx:Panel>
</s:Application>
```

RadialGradient fills

A radial gradient specifies a gradual color transition in the fill color. A radial gradient defines a fill pattern that radiates out from the center of a graphical element.

In FXG, you use the `<RadialGradient>` tag to define a radial gradient. you add a series of GradientEntry objects to the `<RadialGradient>` tag's `entries` Array to define the colors that make up the gradient fill.

In MXML graphics, you use the `<s:RadialGradient>` tag, or the mx.graphics.RadialGradient class. You then define an array of entries, and add to it GradientEntry objects.

FXG example

The following FXG-based example uses a radial gradient in an FXG component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/RadialGradientExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
     <mx:Panel title="Radial Gradient Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
           <s:Group>
                <comps:RadialGradientComp/>
            </s:Group>
     </mx:Panel>
</s:Application>
```

The following FXG file defines the RadialGradientComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/RadialGradientComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Draw rectangle that is filled with a RadialGradient. -->
    <Rect height="100" width="200">
         <stroke>
              <SolidColorStroke color="#000000" weight="2"/>
         </stroke>
         <fill>
              <RadialGradient>
                <GradientEntry color="#0056FF" ratio="0" alpha=".5"/>
                <GradientEntry color="#00CC99" ratio=".33" alpha=".5"/>
                <GradientEntry color="#ECEC21" ratio=".66" alpha=".5"/>
              </RadialGradient>
         </fill>
   </Rect>
</Graphic>
```

MXML graphics example

The following MXML graphics example draws a rectangle and fills it with a RadialGradient fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/RadialGradientExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <mx:Panel title="Radial Gradient MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <s:Rect height="100" width="200">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="2"/>
                 </s:stroke>
                 <s:fill>
                      <s:RadialGradient>
                        <s:GradientEntry color="0x0056FF" ratio="0" alpha=".5"/>
                        <s:GradientEntry color="0x00CC99" ratio=".33" alpha=".5"/>
                        <s:GradientEntry color="0xECEC21" ratio=".66" alpha=".5"/>
                      </s:RadialGradient>
                 </s:fill>
           </s:Rect>
        </s:Group>
     </mx:Panel>
</s:Application>
```

BitmapFill fills

In FXG, you use the `<BitmapFill>` tag to fill an area with the specified bitmap data. In MXML graphics, you use the `<s:BitmapFill>` tag, or the mx.graphics.BitmapFill class.

A bitmap fill can scale or repeat its contents to fit the area. The default behavior is to scale the bitmap data to fill the available area. To prevent scaling, set the fillMode property to clip. If the fill is larger than the available area, the bitmap data is clipped. To scale the bitmap data to the available area, set the `fillMode` property to `scale`. To repeat the bitmap data in the available area, set the `fillMode` property to `repeat`.

When you set the value of the `fillMode` property in ActionScript (for MXML graphics), use the constants defined on the BitmapFillMode class.

You can offset the position of the bitmap data inside the specified fill area by using the `offset` and `origin` position properties. You can rotate the bitmap data with the `rotation` property.

The properties that are supported by the FXG `<BitmapFill>` tag are listed in “Graphics classes and elements” on page 1715.

A fill’s bitmap data is affected by matrix transformations that you can define on the fill.

To specify the source of a bitmap fill, use the `@Embed` directive inside the `source` property. This embeds the source of the fill into the application at compile time.

You cannot specify the source of a bitmap fill at runtime.

FXG example

The following FXG-based example defines a BitmapFill for a rectangle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/FXGBitmapFillExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="BitmapFill Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
           <s:Group>
                <comps:BitmapFillComp/>
            </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the BitmapFillComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/BitmapFillComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
   <!-- Draw rectangle that is filled with a repeating bitmap. -->
   <Rect height="100" width="200">
         <stroke>
              <SolidColorStroke color="#000000" weight="2"/>
         </stroke>
         <fill>
              <BitmapFill
                source="@Embed('../../assets/AirIcon12x12.gif')"
                fillMode="repeat"/>
         </fill>
   </Rect>
</Graphic>
```

MXML graphics example

The following MXML graphics example defines a bitmap fill:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/BitmapFillExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="BitmapFill MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Graphic>
           <!-- Draw rectangle that is filled with a repeating bitmap. -->
           <s:Rect height="100" width="200">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="2"/>
                 </s:stroke>
                 <s:fill>
                      <s:BitmapFill
                        source="@Embed('../assets/AirIcon12x12.gif')"
                        fillMode="repeat"/>
                 </s:fill>
           </s:Rect>
        </s:Graphic>
     </mx:Panel>
</s:Application>
```

For more information, see the BitmapFill class in *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Bitmap graphics in FXG and MXML graphics

A bitmap graphic defines a rectangular region that is filled with bitmap data from a source file.

In FXG, you use the `<BitmapImage>` tag to draw a bitmap graphic. In MXML graphics, you use the `<s:BitmapImage>` tag, or the spark.primitives.BitmapImage class.

A bitmap image can be optionally scaled or repeated to fit the available area. The default behavior is to scale the image to fill the available area. To prevent scaling, set the `fillMode` property to `clip`. If the image is larger than the available area, the fill is clipped. To scale the image to the available area, set the `fillMode` property to `scale`. To repeat the image in the available area, set the `fillMode` property to `repeat`.

The properties that are supported by the FXG `<BitmapImage>` tag are listed in "Graphics classes and elements" on page 1715.

Bitmap graphics support the following types of images:

- PNG
- GIF
- JPG

When used in a clip mask, bitmap graphics are treated as bitmap-filled rectangles. As a result, the alpha channel of the source bitmap is not relevant when it is part of a mask. The bitmap affects the mask in the same manner as the solid filled rectangle.

FXG examples

The following FXG-based example displays bitmap images in FXG:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/BitmapGraphicExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="BitmapGraphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">

        <s:HGroup>
               <!-- Single image, not resized, not repeated. -->
                    <comps:BitmapGraphicComp/>

               <!-- Single image, scaled to fit specified dimensions. -->
                    <comps:ScaledBitmapGraphicComp/>

               <!-- Repeated image to fit specified dimensions. -->
                    <comps:RepeatedBitmapGraphicComp/>
        </s:HGroup>
     </mx:Panel>
</s:Application>
```

The following FXG file defines the BitmapGraphicComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/BitmapGraphicComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <BitmapImage
         height="120"
         width="120"
         fillMode="clip"
         source="@Embed('../../assets/AirIcon12x12.gif')"/>
</Graphic>
```

The following FXG file defines the ScaledBitmapGraphicComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/ScaledBitmapGraphicComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <BitmapImage
         source="@Embed('../../assets/AirIcon12x12.gif')"
         height="120"
         width="120"
         fillMode="scale"
     />
</Graphic>
```

The following FXG file defines the RepeatedBitmapGraphicComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/RepeatedBitmapGraphicComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <BitmapImage
         source="@Embed('../../assets/AirIcon12x12.gif')"
         width="120"
         height="120"
         fillMode="repeat"/>
</Graphic>
```

MXML graphics examples

The following MXML graphics example embeds an image and displays it in three different ways, original size, resized, and tiled:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/BitmapGraphicExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Bitmap MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">

        <s:HGroup>
           <!-- Single image, not resized, not repeated. -->
            <s:Graphic>
                <s:BitmapImage
                     width="120"
                     height="120"
                     fillMode="clip"
                     source="@Embed('../assets/AirIcon12x12.gif')"/>
            </s:Graphic>
           <!-- Single image, scaled to fit specified dimensions. -->
            <s:Graphic>
                <s:BitmapImage
                     source="@Embed('../assets/AirIcon12x12.gif')"
                     width="120"
                     height="120"
                     fillMode="scale"
                 />
            </s:Graphic>
           <!-- Repeated image to fit specified dimensions. -->
            <s:Graphic>
                <s:BitmapImage
                     source="@Embed('../assets/AirIcon12x12.gif')"
                     width="120"
                     height="120"
                     fillMode="repeat"/>
            </s:Graphic>
        </s:HGroup>
     </mx:Panel>
</s:Application>
```

The following MXML graphics example embeds the image once, and reuses that embedded class for each of the BitmapImage instances. It also uses the BitmapFillMode class's constants to define the fill mode for each image.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/BitmapGraphicExampleAS.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">


    <fx:Script>
        <![CDATA[
            import mx.graphics.BitmapFillMode;

            [Embed(source="../assets/AirIcon12x12.gif")]
            [Bindable]
            public var airLogo:Class;

            private function initApp():void {
                clippedImage.fillMode = BitmapFillMode.CLIP;
                repeatedImage.fillMode = BitmapFillMode.REPEAT;
                scaledImage.fillMode = BitmapFillMode.SCALE;
            }
        ]]>
    </fx:Script>

     <mx:Panel title="Bitmap MXML Graphic Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">

        <s:HGroup>
           <!-- Single image, not resized, not repeated. -->
            <s:Graphic>
                <s:BitmapImage id="clippedImage"
                     width="120" height="120"
                     source="{airLogo}"/>
            </s:Graphic>

           <!-- Single image, scaled to fit specified dimensions. -->
            <s:Graphic>
                <s:BitmapImage id="scaledImage"
                     width="120" height="120"
                     source="{airLogo}"/>
            </s:Graphic>

           <!-- Repeated image to fit specified dimensions. -->
            <s:Graphic>
                <s:BitmapImage id="repeatedImage"
                     width="120" height="120"
                     source="{airLogo}"/>
            </s:Graphic>
        </s:HGroup>
     </mx:Panel>
</s:Application>
```

For more information on embedding images in MXML, see "Embedding assets" on page 1699.

For more information on using the BitmapImage class, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Effects in FXG and MXML graphics

You can add effects such as masks, alpha, and blend modes to graphic elements and groups.

Flash Player renders graphic elements and effects on those elements in the following order:

1. GraphicElement or Group
2. Alpha
3. Masks
4. Filters
5. Blend modes

Masks in FXG and MXML graphics

You can add masks to any instance group element. A mask defines the region in which the group's content is rendered. There are two kinds of masking: clipping and alpha.

In *clip masking*, only the actual path, shapes, and fills defined by the mask are used to determine the effect on the source content. Strokes and bitmap filters on the mask are ignored. Filled regions in the mask are considered filled, and Flash Player renders the source content. The type of fill is not relevant: solid color, gradient fill, or bitmap fills in a mask all render the underlying source content, regardless of the alpha values of the mask fill.

In *alpha masking*, the opacity of each pixel in the source content is multiplied by the opacity of the corresponding region of the mask.

To specify a mask, you add a mask as a child of a group or graphic element. A mask must contain a single child tag, either a Group instance or a locally-defined tag from the library. The mask defines its child to be the clipping mask for its parent.

You set the type of masking to use with the `maskType` property on the mask's parent graphical element. You can set the value of this property to either `alpha` or `clip`.

In FXG, if the `<mask>` tag is a child of the root tag, then it must be the first tag. If there is a `<Library>` tag, then it must be the second tag.

FXG example

The following example uses FXG components to define masks:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleMaskExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">
      <s:VGroup gap="10" left="10" top="10">
            <s:RichText text="Ellipse.maskType = 'alpha'"/>
            <comps:AlphaMaskComp/>
            <s:RichText text="Ellipse.maskType = 'clip'"/>
            <comps:ClipMaskComp/>
      </s:VGroup>
</s:Application>
```

The following FXG file defines the AlphaMaskComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/AlphaMaskComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Ellipse width="400" height="200" maskType="alpha">
          <mask>
                <Group>
                      <Rect width="100" height="100">
                            <fill>
                                  <SolidColor alpha="0.1"/>
                            </fill>
                      </Rect>
                </Group>
          </mask>
          <fill>
                <SolidColor color="#FF00FF"/>
          </fill>
    </Ellipse>
</Graphic>
```

The following FXG file defines the ClipMaskComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/ClipMaskComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Ellipse width="400" height="200" maskType="clip">
          <mask>
                <Group>
                      <Rect width="100" height="100">
                            <fill>
                                  <SolidColor/>
                            </fill>
                      </Rect>
                </Group>
          </mask>
          <fill>
                <SolidColor color="#FF00FF"/>
          </fill>
    </Ellipse>
</Graphic>
```

MXML graphics example

The following MXML graphics example shows the two types of masks, and their effect on the underlying graphic:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/SimpleMaskExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
      <s:VGroup gap="10" left="10" top="10">
            <s:RichText text="Ellipse.maskType = 'alpha'"/>
            <s:Ellipse width="400" height="200" maskType="alpha">
                  <s:mask>
                        <s:Group>
                              <s:Rect width="100" height="100">
                                    <s:fill>
                                          <s:SolidColor alpha="0.1"/>
                                    </s:fill>
                              </s:Rect>
                        </s:Group>
                  </s:mask>
                  <s:fill>
                        <s:SolidColor color="#FF00FF"/>
                  </s:fill>
            </s:Ellipse>
            <s:RichText text="Ellipse.maskType = 'clip'"/>
            <s:Ellipse width="400" height="200" maskType="clip">
                  <s:mask>
                        <s:Group>
                              <s:Rect width="100" height="100">
                                    <s:fill>
                                          <s:SolidColor/>
                                    </s:fill>
                              </s:Rect>
                        </s:Group>
                  </s:mask>
                  <s:fill>
                        <s:SolidColor color="#FF00FF"/>
                  </s:fill>
            </s:Ellipse>
      </s:VGroup>
</s:Application>
```

Filters in FXG and MXML graphics

A filter effect consists of a series of graphic operations applied to the rendered graphic of a grouping element and its children before compositing it into the background.

Filters can only be defined on shapes and group instances, not group definitions. That means they can be defined on a Group tag placed as an immediate child of a Graphic or other Group tag, or as a child of a library object that references a defined group.

To define a filter in FXG, add the `<filters>` child tag to the Group or shape you want to apply the filter to. To define a filter in MXML graphics, you add a `<s:filters>` tag to the Group or shape you want to apply the filter to. The filters are defined in the spark.filters package.

The FXG `<filters>` child tag and the `<s:filters>` MXML tag must contain one or more of the bitmap filter types.

Flash Player applies filters to the rendered content of the tag in the order that they appear in the document: the first filter modifies the rendered content, the second filter modifies the output of the first filter, and so on.

The following table lists the filters that are supported by FXG and MXML graphics. It also lists the FXG attributes and children. For the supported properties of the MXML graphics equivalents, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

FXG tag	Equivalent MXML graphics tag and ActionScript class	FXG tag attributes	Children
BevelFilter	<s:BevelFilter> spark.filters.BevelFilter	angle blurX blurY highlightAlpha highlightColor distance knockout quality shadowAlpha shadowColor strength type	None.
BlurFilter	<s:BlurFilter> spark.filters.BlurFilter	blurX blurY quality	None.
ColorMatrixFilter	<s:ColorMatrixFilter> spark.filters.ColorMatrixFilter	matrix	None.
DropShadowFilter	<s:DropShadowFilter> spark.filters.DropShadowFilter	alpha angle blurX blurY color distance inner hideObject knockout quality strength	None.

FXG tag	**Equivalent MXML graphics tag and ActionScript class**	**FXG tag attributes**	**Children**
GlowFilter	<s:GlowFilter> spark.filters.GlowFilter	alpha blurX blurY color inner knockout quality strength	None.
GradientGlowFilter	<s:GradientGlowFilter> spark.filters.GradientGlowFilter	angle blurX blurY distance inner knockout quality strength	GradientEntry
GradientBevelFilter	<s:GradientBevelFilter> spark.filters.GradientBevelFilter	angle blurX blurY distance knockout quality strength type	GradientEntry

MXML graphics also support an AnimateFilter effect, that lets you animate a filter at runtime. Unlike effects that animate properties of the target, the AnimateFilter effect animates properties of the filter applied to the target. For more information, see “Spark filter effects” on page 1823.

In some cases, you can use the RectangularDropShadow class to apply a drop shadow to a graphic element, rather than using the DropShadowFilter effect. It is more efficient, but limited in some ways.

FXG examples

The following FXG-based example applies a drop shadow filter to a rectangle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/FilteredRectExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Rectangle With DropShadowFilter Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:FilteredRectComp/>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the FilteredRectComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/FilteredRectComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
   <Group>
        <filters>
            <DropShadowFilter distance="12" alpha=".5" strength=".33"/>
        </filters>
        <Rect height="100" width="200">
             <stroke>
                  <SolidColorStroke color="#000000" weight="1"/>
             </stroke>
             <fill>
                  <SolidColor color="#FF0000"/>
             </fill>
        </Rect>
   </Group>
</Graphic>
```

You can also apply filters to the RichText element, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/FilteredTextExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="TextGraphic with BlurFilter Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <comps:TextFilterComp/>
        </s:Group>
     </mx:Panel>
</s:Application>
```

The following file defines the TextFilterComp FXG component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/TextFilterComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
   <Group>
        <filters>
            <BlurFilter/>
        </filters>
        <RichText>
           <content>Hello World!</content>
        </RichText>
   </Group>
</Graphic>
```

MXML graphics examples

The following MXML graphics example applies a drop shadow filter to a rectangle:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/FilteredRectExampleMXML.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

     <mx:Panel title="Rectangle With DropShadowFilter MXML Graphics Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
            <s:filters>
                <s:DropShadowFilter distance="12" alpha=".5" strength=".33"/>
            </s:filters>
            <s:Rect height="100" width="200">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="1"/>
                 </s:stroke>
                 <s:fill>
                      <s:SolidColor color="0xFF0000"/>
                 </s:fill>
            </s:Rect>
        </s:Group>
     </mx:Panel>
</s:Application>
```

Blend modes in FXG and MXML graphics

Graphical elements can control how their content is composited with the background by specifying a `blendMode` attribute. A grouping element with the `blendMode` property set to `normal` acts only as a transformation on the elements contained within. Its 2D transform, color transform, and opacity are multiplied down into its child elements, where they are combined with any local values to render the child element into the current surface.

In FXG, you specify a blend mode by setting the value of the `blendMode` attribute on the `<Group>` or graphic element (like `<Rect>`).

In MXML graphics, you specify a blend mode by setting the value of the `blendMode` attribute on the `<s:Group>` tag or graphic tag (like `<s:Rect>`). The possible values are defined by the flash.display.BlendMode class.

The following table describes the available blend modes in FXG:

Blend mode	Description
add	Adds the values of the constituent colors of the element to the colors of its background, applying a ceiling of 0xFF.
alpha	Applies the alpha value of each pixel of the element to the background.
auto	For grouping elements, acts the same as setting the blend mode to normal when the alpha of the group is 0 or 1, and the blend mode is layer when the alpha of the group is strictly between 0 and 1. For shape elements, acts the same as setting the blend mode to normal.
color	Creates a resulting color with the luminance of the base color and the hue and saturation of the blend color. This preserves the gray levels in the artwork and is useful for coloring monochrome artwork and for tinting color artwork.
colordodge	Brightens the base color to reflect the blend color. Blending with black produces no change.
colorburn	Darkens the base color to reflect the blend color. Blending with white produces no change.
darken	Selects the darker of the constituent colors of the element and the colors of the background (the colors with the smaller values).
difference	Compares the constituent colors of the element with the colors of its background, and subtracts the darker of the values of the two constituent colors from the lighter value.
erase	Erases the background based on the alpha value of the element.
exclusion	Creates an effect similar to but lower in contrast than the Difference mode. Blending with white inverts the base-color components. Blending with black produces no change.
hardlight	Adjusts the color of each pixel based on the darkness of the element. If the element is lighter than 50% gray, the element and background colors are screened, which results in a lighter color. If the element is darker than 50% gray, the colors are multiplied, which results in a darker color. This setting is commonly used for shading effects.
hue	Creates a resulting color with the luminance and saturation of the base color and the hue of the blend color.
invert	Inverts the background.
layer	The element content is pre-composed in a temporary buffer before it is processed further.
lighten	Selects the lighter of the constituent colors of the element and the color of the background (the colors with the larger values).
luminosity	Creates a resulting color with the hue and saturation of the base color and the luminance of the blend color. This mode creates an inverse effect from that of the Color mode.
multiply	Multiplies the values of the element's constituent colors by the colors of the background color, and then normalizes by dividing by 0xFF, resulting in darker colors.
normal	The element content appears in front of the background. Pixel values of the element override those of the background. Where the element is transparent, the background is visible.
overlay	Adjusts the color of each pixel based on the darkness of the background. If the background is lighter than 50% gray, the element and background colors are screened, which results in a lighter color. If the background is darker than 50% gray, the colors are multiplied, which results in a darker color.
saturation	Creates a resulting color with the luminance and hue of the base color and the saturation of the blend color. Painting with this mode in an area with no saturation (gray) causes no change.
screen	Multiplies the complement (inverse) of the display object color by the complement of the background color, resulting in a bleaching effect. This setting is commonly used for highlights or to remove black areas of the display object.
softlight	Darkens or lightens the colors, depending on the blend color. The effect is similar to shining a diffused spotlight on the artwork.If the blend color (light source) is lighter than 50% gray, the artwork is lightened, as if it were dodged. If the blend color is darker than 50% gray, the artwork is darkened, as if it were burned in. Painting with pure black or white produces a distinctly darker or lighter area but does not result in pure black or white.
subtract	Subtracts the values of the constituent colors in the element from the values of the background color, applying a floor of 0.

You can also set the blend mode of a shape.

For more information about using blend modes in FXG, see the FXG 2.0 specification.

Color transformations in FXG and MXML graphics

Color transformations are a high performance way to modify the color of a group or shape element. A color transform can be used, for example, to tint a group, adjust its brightness, or modify its opacity.

Color transforms are applied to the rendering of a group or shape element after any bitmap filters are applied; color transforms affect the *output* of a filter (such as tinting the drop shadow). They are applied before the group element is composited with its background. Blend modes use the transformed color value when combining the group with the background content.

In FXG, you specify a color transformation by adding a `<transform>` tag as a child tag of a group. The `<transform>` tag must have a single child tag `<Transform>`. The `<Transform>` tag can have a child tag `<colorTransform>`. The `<colorTransform>` tag must have a single child tag `<ColorTransform>`. The supported attributes of the `<ColorTransform>` tag are shown in "Graphics classes and elements" on page 1715.

In MXML graphics, you specify a color transformation with the flash.geom.ColorTransform class.

Transformations are considered instanced group properties, and can only be defined on shape elements and Groups whose parent tag is another grouping element, or on instances of symbols. Specifically, transformations cannot be defined on Groups whose parent tag is a `<Definition>` tag, or on the topmost `<Graphic>` tag of an FXG document.

Transformations can be defined on an element in one of two ways: through discrete transform attributes, or through a child Transformation and Matrix element. It is illegal to specify both a child element matrix transformation and one or more transform attributes on the same Group instance.

Discrete transforms can be specified with the attributes `x`,`y`, `scaleX`, `scaleY`, and `rotation`. These attributes are combined to create a 2D transform matrix to define the Group's coordinate space as follows:

- Scale by `scaleX`, `scaleY`
- Rotate by `rotation`
- Translate by `x`, `y`

For more information about using color transformations in FXG, see the FXG 2.0 specification.

Introduction to effects

Effects let you add animation to your application in response to user or programmatic action. For example, you can use effects to cause a dialog box to bounce slightly when it receives focus, or to slowly fade in when it becomes visible. You build effects into your applications by using MXML and ActionScript.

About effects

An *effect* is a visible or audible change to the target component that occurs over a time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component.

Effects are initiated in response to an event, where the event is often is initiated by a user action, such as a button click. However, you can initiate effects programmatically or in response to events that are not triggered by the user.

You can define multiple effects to play in response to a single event. For example, when the user clicks a Button control, a window becomes visible. As the window becomes visible, it uses effects to move to the lower-left corner of the screen, and resize itself from 100 by 100 pixels to 300 by 300 pixels.

Flex ships with two types of effects: Spark effects and MX effects. Spark effects are designed to work with all Flex components, including MX components, Spark components, and the Flex graphics components. Because Spark effects can be applied to any component, Adobe recommends that you use the Spark effects in your application when possible.

The MX effects are designed to work with the MX components, and in some cases might work with the Spark components. However, for best results, you should use the Spark effects.

Blogger Brian Telintelo blogged about Flex: Effects on a Mobile Device.

About applying Spark effects

Spark effects are defined in the spark.effects package. To apply a Spark effect, you first define it in the `<fx:Declarations>`, and then invoke the effect by calling the `Effect.play()` method. The following example uses the event listener of a Button control's `click` event to invoke a Resize effect on an Image control:

```
<?xml version="1.0"?>
<!-- behaviors\SparkResizeEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:Resize id="myResizeEffect"
            target="{myImage}"
            widthBy="10" heightBy="10"/>
    </fx:Declarations>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Resize Me"
        click="myResizeEffect.end();myResizeEffect.play();"/>
</s:Application>
```

Notice that this example first calls the `Effect.end()` method before it calls the `play()` method. Call the `end()` method to ensure that any previous instance of the effect has ended before you start a new one.

In the next example, you create two Resize effects for a Button control. One Resize effect expands the size of the button by 10 pixels when the user clicks down on the button, and the second resizes it back to its original size when the user releases the mouse button. The duration of each effect is 200 ms:

```
<?xml version="1.0"?>
<!-- behaviorExamples\SparkResizeEffectReverse.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <s:Resize id="myResizeEffectUp"
            target="{myImage}"
            widthBy="10" heightBy="10"/>
        <s:Resize id="myResizeEffectDown"
            target="{myImage}"
            widthBy="-10" heightBy="-10"/>
    </fx:Declarations>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Resize Me Up"
        click="myResizeEffectUp.end();myResizeEffectUp.play();"/>
    <s:Button label="Resize Me Down"
        click="myResizeEffectDown.end();myResizeEffectDown.play();"/>
</s:Application>
```

About applying MX effects

MX effects are defined in the mx.effects package. With MX effects, you typically use a trigger to initiate the effect. A *trigger* is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. To configure a component to use an effect, you associate an effect with a trigger.

The MX effects are designed to work with the MX components, and in some cases might work with the Spark components.

***Note:** Triggers are not the same as events. For example, a Button control has both a `mouseDown` event and a `mouseDownEffect` trigger. The event initiates the corresponding effect trigger when a user clicks on a component. You use the `mouseDown` event property to specify the event listener that executes when the user clicks on the component. You use the `mouseDownEffect` trigger property to associate an effect with the trigger.*

You associate MX effects with triggers as part of defining the basic behavior for your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWL.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define effect. -->
        <mx:WipeLeft id="myWL" duration="1000"/>
    </fx:Declarations>
    <!-- Assign effect to targets. -->
    <mx:Button id="myButton" label="Click Me"
        mouseDownEffect="{myWL}"/>
    <mx:Button id="myOtherButton" label="Click Me"
        mouseDownEffect="{myWL}"/>
</s:Application>
```

In this example, the effect is a WipeLeft effect with a duration of 1000 milliseconds (ms). That means it takes 1000 ms for the effect to play from start to finish.

You use data binding to assign the effect to the `mouseDownEffect` property of each Button control. The `mouseDownEffect` property is the effect trigger that specifies to play the effect when the user clicks the control using the mouse. In the previous example, the effect makes the Button control appear as if it is being wiped onto the screen from right to left.

About factory and instance classes

Flex implements effects using an architecture in which each effect is represented by two classes: a factory class and an instance class.

Factory class You use factory classes in your application. The factory class creates an object of the instance class to perform the effect on the target. The factory classes are defined in the spark.effects and mx.effects packages.

You define a factory class in your application, and configure it with the necessary properties to control the effect, such as the zoom size or effect duration, as the following example shows:

```
<fx:Declarations>
    <!-- Factory class. -->
    <s:Resize id="myResizeEffect"
        target="{myImage}"
        widthBy="50" heightBy="50"/>
</fx:Declarations>
<!-- Effect target.-->
<s:Image id="myImage"/>
<s:Button label="Resize Image"
    click="myResizeEffect.end();myResizeEffect.play();"/>
```

By convention, the name of a factory class is the name of the effect, such as Resize, Move, or Fade.

Instance class The instance class implements the effect logic. When an effect plays, the factory class creates an object of the instance class to perform the effect on the target. When the effect ends, Flex destroys the instance object. If the effect has multiple target components, the factory class creates multiple instance objects, one per target.

The factory classes are defined in the spark.effects.supportClasses and mx.effects.effectClasses packages. By convention, the name of an instance class is the name of the factory class with the suffix *Instance*, such as ResizeInstance, MoveInstance, or FadeInstance.

When you use effects, you perform the following steps.

1 Create a factory class object in your application.

2 Configure the factory class object.

When Flex plays an effect, Flex performs the following actions:

1 Creates one object of the instance class for each target component of the effect.

2 Copies configuration information from the factory object to the instance object.

3 Plays the effect on the target using the instance object.

4 Deletes the instance object when the effect completes.

Any changes that you make to the factory object are not propagated to a currently playing instance object. However, the next time the effect plays, the instance object uses your new settings.

When you use effects in your application, you are concerned only with the factory class; the instance class is an implementation detail. However, if you want to create custom effects classes, you must implement a factory and an instance class.

For more information, see “Custom effects” on page 2525.

Available effects

The following table lists the Spark and MX effects:

Spark Effect	MX Effect	Description

`Animate`
`Animae3D`
`AnimateColor`
	`AnimateProperty`	Animates a numeric property of a component, such as `height`, `width`, `scaleX`, or `scaleY`. You specify the property name, start value, and end value of the property to animate. The effect first sets the property to the start value, and then updates the property value over the duration of the effect until it reaches the end value.
		For example, if you want to change the width of a Button control, you can specify width as the property to animate, and starting and ending width values to the effect.

`AnimateFilter`		Animates the properties of a filter applied to the target. This effect does not actually modify the properties of the target. For example, you can use this property to animate a filter, such as a DropShadowFilter, applied to the target.
	`Blur`	Applies a blur visual effect to a component. A Blur effect softens the details of an image. You can produce blurs that range from a softly unfocused look to a Gaussian blur, a hazy appearance like viewing an image through semi-opaque glass. If you apply a Blur effect to a component, you cannot apply a BlurFilter or a second Blur effect to the component. The Blur effect uses the Flash BlurFilter class as part of its implementation. For more information, see flash.filters.BlurFilter in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. You can use the Spark AnimateFilter effect to replace the MX Blur effect.
`CallAction`		Animates the target by calling a function on the target. The effect lets you pass parameters to the function from the effect class.

Spark Effect	MX Effect	Description
	`Dissolve`	Modifies the `alpha` property of an overlay to gradually have to target component appear or disappear. If the target object is a container, only the children of the container dissolve. The container borders do not dissolve. Note: To use the MX Dissolve effect with the `creationCompleteEffect` trigger of a DataGrid control, you must define the data provider of the control inline using a child tag of the DataGrid control, or using data binding. This issue is a result of the data provider not being set until the `creationComplete` event is dispatched. Therefore, when the effect starts playing, Flex has not completed the sizing of the DataGrid control.
`Fade` `CrossFade`	`Fade`	Animate the component from transparent to opaque, or from opaque to transparent. If you specify the MX `Fade` effect for the `showEffect` or `hideEffect` trigger, and if you omit the `alphaFrom` and `alphaTo` properties, the effect automatically transitions from 0.0 to the targets' current `alpha` value for a show trigger, and from the targets' current `alpha` value to 0.0 for a hide trigger. Note: To use these effects with MX components that display text, the component must either support the Flash Text Engine or you must use an embedded font with the component, not a device font. For more information, see "Styles and themes" on page 1492.
	`Glow`	Applies a glow visual effect to a component. The Glow effect uses the Flash GlowFilter class as part of its implementation. For more information, see the flash.filters.GlowFilter class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. If you apply a Glow effect to a component, you cannot apply a GlowFilter or a second Glow effect to the component. You can use the Spark AnimateFilter effect to replace the MX Glow effect.
	`Iris`	Animates the effect target by expanding or contracting a rectangular mask centered on the target. The effect can either grow the mask from the center of the target to expose the target, or contract it toward the target center to obscure the component. For more information, see "Using MX mask effects" on page 1837.
`Move`	`Move`	Changes the position of a component over a specified time interval. You typically apply this effect to a target in a container that uses absolute positioning, such as a MX Canvas container, or a Spark container that uses BasicLayout. If you apply it to a target in a container that performs automatic layout, the move occurs, but the next time the container updates its layout, it moves the target back to its original position. You can set the container's `autoLayout` property to `false` to disable the move back, but that disables layout for all controls in the container.
	`Pause`	Does nothing for a specified period of time. This effect is useful when you need to composite effects. For more information, see "Creating composite effects" on page 1799.
`Resize`	`Resize`	Changes the width and height of a component over a specified time interval. When you apply a Resize effect, the layout manager resizes neighboring components based on the size changes to the target component. To run the effect without resizing other components, place the target component in a Canvas container, or a Spark container that uses BasicLayout. When you use the Resize effect with Panel containers, you can hide Panel children to improve performance. For more information, see "Improving performance when resizing Panel containers" on page 1842.

Spark Effect	**MX Effect**	**Description**
`Rotate` `Rotate3D`	`Rotate`	Rotates a component around a specified point. You can specify the coordinates of the center of the rotation, and the starting and ending angles of rotation. You can specify positive or negative values for the angles. Note: To use the rotate effects with MX components that display text, the component must either support the Flash Text Engine or you must use an embedded font with the component, not a device font. For more information, see "Styles and themes" on page 1492.
`Scale Scale3D`		Scale a component. You can specify properties to scale the target in the x and y directions.
	`SoundEffect`	Plays an mp3 audio file. For example, you could play a sound when a user clicks a Button control. This effect lets you repeat the sound, select the source file, and control the volume and pan. You specify the mp3 file using the `source` property. If you have already embedded the mp3 file, using the `Embed` keyword, then you can pass the Class object of the mp3 file to the source property. Otherwise, specify the full URL to the mp3 file. For more information, see "Using the MX sound effect" on page 1836.

`Wipe`	`WipeLeft`
	`WipeRight`
	`WipeUp`
	`WipeDown`
		Defines a bar Wipe effect. The before or after state of the component must be invisible.

	`Zoom`	Zooms a component in or out from its center point by scaling the component. Note: When you apply a Zoom effect to text rendered using a system font, Flex scales the text between whole point sizes. Although you do not have to use embedded fonts when you apply a Zoom effect to text, the Zoom will appear smoother when you apply it to embedded fonts. For more information, see "Styles and themes" on page 1492. You can use the Spark Scale effect to replace the MX Zoom effect.

Applying effects

To apply an effect to a target, you must specify the target, and then initiate the effect. Once the effect has started, you can pause, stop, resume, and end the effect epigrammatically.

Using the Effect.target and Effect.targets properties

Use the Effect.target or Effect.targets properties to specify the effect targets. You use the `Effect.target` property in MXML to specify a single target, and the `Effect.targets` property to specify an array of targets, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\TargetProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Resize id="myResize"
            heightBy="25"
            widthBy="50"
            target="{myButton}"/>
    </fx:Declarations>

    <s:Button id="myButton"
        label="Resize target"
        click="myResize.end();myResize.play();"/>
</s:Application>
```

In this example, you use data binding to the `target` property to specify that the Button control is the target of the Resize effect.

In the next example, you apply a Resize effect to multiple Button controls by using data binding with the effect's `targets` property:

```
<?xml version="1.0"?>
<!-- behaviors\TargetProp3Buttons.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <s:Resize id="myResize"
            heightBy="25"
            widthBy="50"
            targets="{[myButton1, myButton2, myButton3]}"/>
    </fx:Declarations>
    <s:Button id="myButton1" label="Button 1"/>
    <s:Button id="myButton2" label="Button 2"/>
    <s:Button id="myButton3" label="Button 3"/>

    <s:Button id="myButton4"
        label="Zoom targets"
        click="myResize.end();myResize.play();"/>
</s:Application>
```

Because you specified three targets to the effect, the `play()` method invokes the effect on all three button controls.

If you use the `targets` property to define multiple event targets, calling the `end()` method with no arguments terminates the effect on all targets. If you pass an effect instance as an argument, just that instance is interrupted.

To obtain the effect instance, save the return value of the `play()` method, as the following example shows:

```
var myResizeArray:Array = myResize.play();
```

The Array contains EffectInstance objects, one per target of the effect. Pass the element from the Array to the `end()` method that corresponds to the effect to end, as the following example shows:

```
myResize.end(myResizeArray[1]);
```

You can define an effect that specifies no targets. Instead, you can pass an Array of targets to the `play()` method to invoke the effect on all components specified in the Array, as the following example shows:

```
myResize.play([comp1, comp2, comp3]);
```

This example invokes the Resize effect on three components.

Applying effects by using data binding

You can use data binding in MXML to set properties of an effect. For example, the following example lets the user set the `heightBy` and `widthBy` properties of the Resize effect using a TextInput control. The `heightBy` and `widthBy` properties specify the increase, of pixels, of the size of the target.

```
<?xml version="1.0"?>
<!-- behaviors\DatabindingEffects.mxml  -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:Resize id="myResizeEffect"
            target="{myImage}"
            heightBy="{Number(resizeHeightInput.text)}"
            widthBy="{Number(resizeWidthInput.text)}"/>
    </fx:Declarations>
    <s:Form>
        <s:FormItem label="Resize Height:">
            <s:TextInput id="resizeHeightInput"
                text="0" width="30"/>
        </s:FormItem>
        <s:FormItem label="Resize Width:">
            <s:TextInput id="resizeWidthInput"
                text="0" width="30"/>
        </s:FormItem>
    </s:Form>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Resize Image" click="myResizeEffect.play();"/>
</s:Application>
```

Playing an effect backward

You can pass an optional argument to the `play()` method to play the effect backward, as the following example shows:

```
resizeLarge.play([comp1, comp2, comp3], true);
```

In this example, you specify `true` as the second argument to play the effect backward. The default value is `false`.

You can also use the Effect.pause() method to pause an effect, the Effect.resume() method to resume a paused effect, and the Effect.reverse() method to play an effect backward.

Ending an effect

Use the end() method to terminate an effect at any time, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ASend.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Resize id="resizeLarge"
            heightTo="150"
            widthTo="150"
            duration="5000"
            target="{myTA}"/>
    </fx:Declarations>

    <s:Group height="228" width="328">
        <s:Button label="Start"
            x="10" y="10"
            click="resizeLarge.end();resizeLarge.play();"/>
        <s:Button label="End"
            x="86" y="10"
            click="resizeLarge.end();"/>
        <s:Button label="Reset"
            click="myTA.height=100;myTA.width=100;"
            x="162" y="10"/>
        <s:TextArea id="myTA"
            x="10" y="40"
            height="100"
            width="100"
            text="Here is some text."/>
    </s:Group>
</s:Application>
```

In this example, you set the `duration` property of the Resize effect to 10 seconds, and use a Button control to call the `end()` method to terminate the effect when the user clicks the button.

When you call the `end()` method, the effect jumps to its end state and then terminates. For the Resize effect, the effect sets the final size of the expanded TextArea control before it terminates, just as if you had let the effect finish playing. If the effect was a move effect, the target component moves to its final position before terminating.

You can end all effects on a component by calling the `UIComponent.endEffectsStarted()` method on the component. The `endEffectsStarted()` method calls the `end()` method on every effect currently playing on the component.

If you defined a listener for the `effectEnd` event, that listener gets invoked by the `end()` method, just as if you had let the effect finish playing. For more information on working with effect events, see "Handling effect events" on page 1796.

After the effect starts, you can use the `pause()` method to pause the effect at its current location. You can then call the `resume()` method to start the effect, or the `end()` method to terminate it.

Use the stop() method to stop the effect in its current state, but not jump to the end. A call to the `stop()` method dispatches the `effectEnd` event. Unlike a call to the `pause()` method, you cannot call the `resume()` method after calling the `stop()` method. However, you can call the `play()` method to restart the effect.

Creating effects in ActionScript

You can declare and play effects in ActionScript. The following example uses the event listener of a Button control's `click` event to invoke a Resize effect on an TextArea control:

```
<?xml version="1.0"?>
<!-- behaviors\ASplayVBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="createEffect(event);">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Import effect class.
            import spark.effects.Resize;
            // Create a resize effect
            private var resizeLarge:Resize = new Resize();
            private function createEffect(eventObj:Event):void {
                // Set the TextArea as the effect target.
                resizeLarge.target=myTA;

                // Set resized width and height, and effect duration.
                resizeLarge.widthTo=150;
                resizeLarge.heightTo=150;
                resizeLarge.duration=750;
            }
        ]]>
    </fx:Script>
    <s:Button label="Start"
        click="resizeLarge.end();resizeLarge.play();"/>
    <s:Button label="Reset"
        click="myTA.width=100;myTA.height=100;"/>
    <s:TextArea id="myTA"
        height="100" width="100"
        text="Here is some text."/>
</s:Application>
```

In this example, use the application's `creationComplete` event to configure the effect, and then invoke it by calling the `play()` method in response to a user clicking the Button control.

Working with effects

Effects have many configuration settings that you can use to control the effect. For example, you can set the effect duration and repeat behavior, or handle effect events. The effect target also has configuration settings that you can use to configure it for effects.

Setting effect durations

All effects take the `duration` property that you can use to specify the time, in milliseconds, over which the effect occurs. The following example creates two versions of the Fade effect. The slowFade effect uses a two-second duration; the reallySlowFade effect uses an eight-second duration:

```
<?xml version="1.0"?>
<!-- behaviors\FadeDuration.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:Fade id="slowFade"
            duration="2000"
            target="{myButton1}"/>
        <s:Fade id="reallySlowFade"
            duration="8000"
            target="{myButton2}"/>
    </fx:Declarations>
    <s:Button id="myButton1"
        label="Button 1"
        creationCompleteEffect="{slowFade}"/>
    <s:Button id="myButton2"
        label="Button 2"
        creationCompleteEffect="{reallySlowFade}"/>
</s:Application>
```

Delaying effect start

The Effect.startDelay property specifies a value, in milliseconds, that the effect waits once it is triggered before it begins. You can specify an integer value greater than or equal to 0. If you have used the `Effect.repeatCount` property to specify the number of times to repeat the effect, the `startDelay` property is applied only to the first time the effect plays, but not to the repeated playing of the effect.

If you set the `startDelay` property for a Parallel effect, Flex inserts the delay between each effect of the parallel effect.

Repeating effects

All effects support the Effect.repeatCount and Effect.repeatDelay properties that let you configure whether effects repeat, where:

- `repeatCount` Specifies the number of times to play the effect. A value of 0 means to play the effect indefinitely until stopped by a call to the `end()` method. The default value is 1. For a repeated effect, the `duration` property specifies the duration of a single instance of the effect. Therefore, if an effect has a `duration` property set to 2000, and a `repeatCount` property set to 3, then the effect takes a total of 6000 ms (6 seconds) to play.
- `repeatDelay` Specifies the amount of time, in milliseconds, to pause before repeating the effect. The default value is 0.
- `repeatBehavior` (Spark effects only) Specifies `RepeatBehavior.LOOP` (default) to repeat the effect each time, or `RepeatBehavior.REVERSE` to reverse direction of the effect on each iteration.

For example, the following example repeats the Rotate effect until the user clicks a Button control:

```
<?xml version="1.0"?>
<!-- behaviors\RepeatEff.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <s:Rotate id="myRotate"
            angleBy="360"
            repeatCount="0"
            target="{myImage}"/>
    </fx:Declarations>
    <s:Label text="Click the image to start rotation."/>
    <s:Button id="myButton"
        label="Stop Rotation"
        click="myRotate.end();"/>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"
        mouseDown="myRotate.end(); myRotate.play();"/>
</s:Application>
```

All effects dispatch an `effectEnd` event when the effect completes. If you repeat the effect, the effect dispatches the `effectEnd` event after the final repetition.

If the effect is a tween effect, such as a MX Fade or MX Move effect, the effect dispatches both the `tweenEnd` effect and the `endEffect` when the effect completes. If you configure the tween effect to repeat, the `tweenEnd` effect occurs at the end of every repetition of the effect, and the `endEffect` event occurs after the final repetition.

Handling effect events

Every Spark and MX effect class supports the following events:

- effectStart Dispatched when the effect starts playing. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_START`.
- effectEnd Dispatched after the effect ends, either when the effect finishes playing or when the effect has been interrupted by a call to the `end()` method. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_END`.
- effectStop Dispatched after the effect stops by a call to the `stop()` method. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_STOP`.

The event object passed to the event listener for these events is of type EffectEvent.

The previous list of events are dispatched by all effects. The Spark and MX effects dispatch additional events. For more information, see “Handling Spark effect events” on page 1826 and “Handling MX effect events” on page 1835.

Flex dispatches one event for each target of an effect. Therefore, if you define a single target for an effect, Flex dispatches a single `effectStart` event, and a single `effectEnd` event. If you define three targets for an effect, Flex dispatches three `effectStart` events, and three `effectEnd` events.

The EffectEvent class is a subclass of the Event class, and contains all of the properties inherited from Event, including `target`, and `type`, and defines a new property named `effectInstance`, where:

target Contains a reference to the Effect object that dispatched the event. This is the factory class of the effect.

type Either `EffectEvent.EFFECT_END` or `EffectEvent.EFFECT_START`, depending on the event.

effectInstance Contains a reference to the EffectInstance object. This is the object defined by the instance class for the effect. Flex creates one object of the instance class for each target of the effect. You access the target component of the effect using the `effectInstance.target` property.

The following example defines an event listener for the `endEffect` event:

```
<?xml version="1.0"?>
<!-- behaviors\EventEffects2.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            import mx.effects.*;
            import mx.events.EffectEvent;
            import mx.core.UIComponent;

            private function endEffectListener(eventObj:EffectEvent):void {
                // Access the effect object.
                var effectObj:Effect = Effect(eventObj.target);
                // Access the target component of the effect.
                var effectTarget:UIComponent =
                    UIComponent(eventObj.effectInstance.target);
                // Write the target id and event type to the TextArea control.
                myTA.text = effectTarget.id;
                myTA.text = myTA.text + " " + eventObj.type;
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:Fade id="slowFade"
            duration="2000"
            effectEnd="endEffectListener(event);"
            target="{myButton1}"/>
    </fx:Declarations>
    <s:Button id="myButton1"
        label="Button 1"
        creationCompleteEffect="{slowFade}"/>
    <s:TextArea id="myTA" />
</s:Application>
```

If the effect has multiple targets, Flex dispatches an `effectStart` event and `effectEnd` event once per target, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\EventEffects.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            import mx.effects.*;
            import mx.events.EffectEvent;
            import mx.core.UIComponent;

            private function endSlowFadeEffectListener(eventObj:EffectEvent):void
            {
                // Access the effect object.
                var effectObj:Effect = Effect(eventObj.target);
                // Access the target component of the effect.
                var effectTarget:UIComponent =
                    UIComponent(eventObj.effectInstance.target);

                myTA.text = myTA.text + effectTarget.id + ' : ';
                myTA.text = myTA.text + " " + eventObj.type + '\n';
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <s:Fade id="slowFade"
            duration="2000"
            effectEnd="endSlowFadeEffectListener(event);"
            targets="{[myButton1, myButton2, myButton3, myButton4]}"/>
    </fx:Declarations>

    <s:Button id="myButton1"
        label="Button 1"
        creationCompleteEffect="{slowFade}"/>
    <s:Button id="myButton2"
        label="Button 2"
        creationCompleteEffect="{slowFade}"/>
    <s:Button id="myButton3"
        label="Button 3"
        creationCompleteEffect="{slowFade}"/>
    <s:Button id="myButton4"
        label="Button 4"
        creationCompleteEffect="{slowFade}"/>

    <s:TextArea id="myTA" height="125" width="250"/>
</s:Application>
```

Flex dispatches an `effectEnd` event once per target; therefore, the `endSlowFadeEffectListener()` event listener is invoked four times, once per Button control.

Creating composite effects

Flex supports two ways to combine, or *composite*, effects:

Parallel The effects play at the same time. If you play effects in parallel, you must make sure that the effects do not modify the same property of the target. If two effects modify the same property, the effects conflict with each other and the results of the effects are undefined.

Sequence One effect must complete before the next effect starts.

To define a Parallel or Sequence effect, you use the `<mx:Parallel>` or `<,x:Sequence>` tag. The following example defines the Parallel effect, fadeResizeShow, which combines the Spark Fade and Resize effects in parallel, and fadeResizeHide, which combines the Fade and Resize effects in sequence:

```
<?xml version="1.0"?>
<!-- behaviors\CompositeEffects.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <s:Sequence id="fadeResizeHide"
            target="{aTextArea}"
            duration="1000">
            <s:Fade id="fadeHide"
                alphaFrom="1.0"
                alphaTo="0.0"/>
            <s:Resize id="resizeHide"
                widthTo="0"
                heightTo="0"/>
        </s:Sequence>
        <s:Parallel id="fadeResizeShow"
            target="{aTextArea}"
            duration="1000">
            <s:Resize id="resizeShow"
                widthTo="100"
                heightTo="50"/>
            <s:Fade id="fadeShow"
                alphaFrom="0.0"
                alphaTo="1.0"/>
        </s:Parallel>
    </fx:Declarations>
    <s:TextArea id="aTextArea"
        width="100" height="50"
        text="Hello world."/>
    <s:Button id="myButton2"
        label="Hide!"
        click="fadeResizeHide.end();fadeResizeHide.play();"/>
    <s:Button id="myButton1"
        label="Show!"
        click="fadeResizeShow.end();fadeResizeShow.play();"/>
</s:Application>
```

The button controls alternates making the TextArea control visible and invisible. When the TextArea control becomes invisible, it uses the fadeResizeHide effect as its hide effect, and when it becomes invisible, it uses the fadeResizeShow effect.

As a modification to this example, you could disable the Show button when the TextArea control is visible, and disable the Hide button when the TextArea control is invisible, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\CompositeEffectsEnable.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            private function showHandler():void {
                myButton2.enabled=true;
                myButton1.enabled=false;
            }
            private function hideHandler():void {
                myButton2.enabled=false;
                myButton1.enabled=true;
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:Sequence id="fadeResizeHide"
            target="{aTextArea}"
            duration="1000"
            effectEnd="hideHandler();">
            <s:Fade id="fadeHide"
                alphaFrom="1.0"
                alphaTo="0.0"/>
            <s:Resize id="resizeHide"
                widthTo="0"
                heightTo="0"/>
        </s:Sequence>
        <s:Parallel id="fadeResizeShow"
            target="{aTextArea}"
            duration="1000"
            effectEnd="showHandler();">
            <s:Resize id="resizeShow"
                widthTo="100"
                heightTo="50"/>
            <s:Fade id="fadeShow"
                alphaFrom="0.0"
                alphaTo="1.0"/>
        </s:Parallel>
    </fx:Declarations>
    <s:TextArea id="aTextArea"
        width="100" height="50"
        text="Hello world."/>
    <s:Button id="myButton2"
        label="Hide!"
        click="fadeResizeHide.end();fadeResizeHide.play();"/>
    <s:Button id="myButton1"
        label="Show!"
        enabled="false"
        click="fadeResizeShow.end();fadeResizeShow.play();"/>
</s:Application>
```

In this example, the Show button is initially disabled. Event handlers for the `effectEnd` event toggle the `enable` property for the two button based on the effect that played.

You can nest `<Parallel>` and `<Sequence>` tags inside each other. For example, two effects can run in parallel, followed by a third effect running in sequence.

In a Parallel or Sequence effect, the `duration` property sets the duration of each effect. For example, if the a Sequence effect has its `duration` property set to 3000, then each effect in the Sequence will take 3000 ms to play.

Using embedded fonts with effects

The fade and rotate effects only work with components that support the Flash Text Engine (FTE), or with components that use an embedded font. All Spark components, and some MX components, support the FTE. Therefore, the fade and rotate effects work with text in the components.

However, if you apply a fade and rotate effects to a MX component that uses a system font, nothing happens to the text in the component. You either have to embed a font, or use the appropriate Spark component instead of the MX component.

When you apply an MX Zoom effect to text rendered using a system font, Flex scales the text between whole point sizes. While you do not have to use embedded fonts when you apply a Zoom effect to text, the Zoom will appear smoother when you apply it to embedded fonts.

The following example rotates text area controls. The first text area control is defined using the Spark TextArea component, whish supports the FTE. Therefore the text rotates when you apply the Spark Rotate effect to it.

The second text area control is defined using the MX TextArea control. This TextArea control uses an embedded font and, therefore, the text rotates.

The third text area control uses the MX TextArea control with the default system font. Therefore, when you apply the Spark Rotate effect, the text disappears and reappears when you rotate the component back to its initial state:

```
<?xml version="1.0"?>
<!-- behaviors\EmbedFont.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="650">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        @font-face {
            src:url("../assets/MyriadWebPro.ttf");
            font-family: myMyriadWebPro;
            embedAsCFF: true;
        }
        @font-face {
            src:url("../assets/MyriadWebPro.ttf");
            font-family: myMyriadWebProMX;
            embedAsCFF: false;
        }
</fx:Style>
    <fx:Declarations>
        <s:Rotate id="rotateForward"
            angleFrom="0" angleTo="45"/>
        <s:Rotate id="rotateBack"
            angleFrom="45" angleTo="0"/>
    </fx:Declarations>
    <s:VGroup>
        <s:Button label="Rotate Forward"
            click="rotateForward.end();rotateForward.play([l1]);"/>
        <s:Button label="Rotate Backward"
            click="rotateBack.end();rotateBack.play([l1]);"/>
        <s:TextArea id="l1" height="75"
            fontFamily="myMyriadWebPro"
            text="FTE supported. This text will rotate."/>
    </s:VGroup>
    <s:VGroup>
        <s:Button label="Rotate Forward"
            click="rotateForward.end();rotateForward.play([l2]);"/>
        <s:Button label="Rotate Backward"
            click="rotateBack.end();rotateBack.play([l2]);"/>
        <mx:TextArea id="l2" height="75"
            fontFamily="myMyriadWebProMX"
            text="Embedded font. This text will rotate."/>
    </s:VGroup>
    <s:VGroup>
        <s:Button label="Rotate Forward"
            click="rotateForward.end();rotateForward.play([l3]);"/>
        <s:Button label="Rotate Backward"
            click="rotateBack.end();rotateBack.play([l3]);"/>
        <mx:TextArea id="l3" height="75"
            text="System font. This text will not rotate."/>
    </s:VGroup>
</s:Application>
```

Notice that you had to embed the font twice: once for the Spark TextArea control and one for the MX TextArea control. For more information on embedding fonts for MX components, see "Embedding fonts with MX components" on page 1592.

Suspending background processing

To improve the performance of effects, you can disable background processing in your application for the duration of the effect by setting the Effect.suspendBackgroundProcessing property to `true`. The background processing that is blocked includes component measurement and layout, and responses to data services for the duration of the effect.

The default value of the `suspendBackgroundProcessing` property is `false`. You can set it to `true` in most cases. However, you should set it to `false` if either of the following conditions is true for your application:

- User input may arrive while the effect is playing, and the application must respond to the user input before the effect finishes playing.
- A response may arrive from the server while the effect is playing, and the application must process the response while the effect is still playing.

Disabling container layout for effects

By default, Flex updates the layout of a container's children when a new child is added to it, when a child is removed from it, when a child is resized, and when a child is moved. Because some effects, such as the move and resize effects, modify a child's position or size, they cause the container to update its layout.

However, when the container updates its layout, it can actually reverse the results of the effect. For example, you use a move effect to reposition a container child. At some time later, you change the size of another container child, which forces the container to update its layout. This layout update can cause the child that moved to be returned to its original position.

To prevent Flex from performing layout updates, you can set the autoLayout property of a container to `false`. Its default value is `true`, which configures Flex so that it always updates layouts. You always set the `autoLayout` property on the parent container of the component that uses the effect. For example, if you want to control the layout of a child of a Grid container, you set the `autoLayout` property for the parent GridItem container of the child, not for the Grid container.

You set the `autoLayout` property to `false` when you use a move effect in parallel with a resize or zoom effect. You must do this because the resize or zoom effect can cause an update to the container's layout, which can return the child to its original location.

When you use the Zoom effect on its own, you can set the `autoLayout` property to `false`, or you may leave it with its default value of `true`. For example, if you use a Zoom effect with the `autoLayout` property set to `true`, as the child grows or shrinks, Flex automatically updates the layout of the container to reposition its children based on the new size of the child. If you use a Zoom effect with the `autoLayout` property set to `false`, the child resizes around its center point, and the remaining children do not change position.

The container in the following example uses the default vertical alignment of `top` and the default horizontal alignment of `left`. If you apply a Zoom effect to the image, the container resizes to hold the image, and the image remains aligned with the upper-left corner of the container:

```
<s:SkinnableContainer>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Image source="myImage.jpg"/>
</s:SkinnableContainer>
```

In the next example, the image is centered in the container. If you apply a Zoom effect to the image, as it resizes, it remains centered in the container.

```
<s:SkinnableContainer>
    <s:layout>
        <s:VerticalLayout horizontalAlign="center"/>
    </s:layout>
    <s:Image source="myImage.jpg"/>
</s:SkinnableContainer>
```

By default, the size of the container is big enough to hold the image at it original size. If you disable layout updates, and use the Zoom effect to enlarge the image, or use a move effect to reposition the image, the image might extend past the boundaries of the container, as the following example shows:

```
<s:SkinnableContainer autoLayout="false">
    <s:layout>
        <s:VerticalLayout horizontalAlign="center"/>
    </s:layout>
    <s:Image source="myImage.jpg"/>
</s:SkinnableContainer>
```

For a Spark container, if you set the `autoLayout` property to `false`, the container does not resize as the image resizes. The image can grow to a size so that it extends beyond the boundaries of the container. You can then decide to wrap the container in the Scroller component to add scroll bars rather than allowing the Image to extend past the container boundaries.

For an MX container, if you set the `autoLayout` property to `false`, the container does not resize as the image resizes. If the image grows to a size larger than the boundaries of the container, the container adds scroll bars and clips the image at its boundaries.

Setting UIComponent.cachePolicy on the effect target

An effect can use the bitmap caching feature in Adobe® Flash® Player to speed up animations. An effect typically uses bitmap caching when the target component’s drawing does not change while the effect is playing.

For example, the Fade effect works by modifying the `alpha` property of the target component. Changing the `alpha` property does not change the way the target component is drawn on the screen. Therefore, caching the target component as a bitmap can speed up the performance of the effect. The Move effect modifies the `x` and `y` properties of the target component. Modifying the values of these properties does not alter the way the target component is drawn, so it can take advantage of bitmap caching.

Not all effects can use bitmap caching. Effects such as Zoom, resize, and the wipe effects modify the target component in a way that alters the way it is drawn on the screen. The Zoom effect modifies the scale properties of the component, which changes its size. Caching the target component as a bitmap for such an effect would be counterproductive because the bitmap changes continuously while the effect plays.

The UIComponent.cachePolicy property controls the caching operation of a component during an effect. The `cachePolicy` property can have the following values:

CachePolicy.ON Specifies that the effect target is always cached.

CachePolicy.OFF Specifies that the effect target is never cached.

CachePolicy.AUTO Specifies that Flex determines whether the effect target should be cached. This is the default value.

Flex uses the following rules to set the `cacheAsBitmap` property:

- When at least one effect that does not support bitmap caching is playing on a target, set the target's `cacheAsBitmap` property to `false`.

- When one or more effects that supports bitmap caching are playing on a target, set the target's `cacheAsBitmap` property to `true`.

  Typically, you leave the `cachePolicy` property with its default value of `CachePolicy.AUTO`. However, you might want to set the property to `CachePolicy.OFF` because bitmap caching is interfering with your user interface, or because you know something about your application’s behavior such that disabling bitmap caching will have a beneficial effect on it.

Spark effects

Spark effects define a change to the target over time. The Spark effects ship in the spark.effects package.

While Flex ships with both Spark and MX effects, Adobe recommends that you use the Spark effects when possible.

For an introduction to effects, see “Introduction to effects” on page 1784.

About Spark effects

The Spark effects are divided into categories based on their implementation and target type, as the following table shows:

Type	Description
Property effects	Animates the change of one or more properties of the target.
Transform effects	Animates the change of one or more transform-related properties of the target, such as the scale, rotation, and position. The target can be modified in parallel with other transform effects with no interference among the effects.
Pixel-shader effects	Animates the change from one bitmap image to another, where the bitmap image represents the before and after states of the target.
Filter effects	Applies a filter to the target where the effect modifies the properties of the filter, not properties of the target.
3D effects	A subset of the transform effects that modify the 3D transform properties of the target.

Spark effects can be applied to:

- Any Spark or MX component.
- Any graphical component in the spark.primitives package, such as Rect, Ellipse, and path.
- Any object that contains the styles or properties modified by the effect.

Blogger Brian Telintelo blogged about Flex: Effects on a Mobile Device.

Spark property effects

The Spark property effects modify one or more properties of a component over time to animate the effect. The Spark property effects are subclasses of the Animate class, and include the following classes:

- Animate   Animates any properties of the target.
- Fade   Animates a change to the `alpha` property of the target.
- Resize   Animates a change to the `height` and `width` properties of the target.
- AnimateColor   Animates a change to any color property of the target, such as `color`, or `fontColor`.

The Animate effect

Use the Animate effect directly to animate any properties of the target. You create an instance of the SimpleMotionPath class for each property to animate, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkAnimateProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Animate id="scaleUp"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.0" valueTo="1.5"/>
        </s:Animate>
        <s:Animate id="scaleDown"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.5" valueTo="1.0"/>
        </s:Animate>
    </fx:Declarations>
    <s:Button id="myB1"
        label="Scale Button"
        mouseDown="scaleUp.end(); scaleUp.play();"
        mouseUp="scaleDown.end(); scaleDown.play();"/>
</s:Application>
```

The SimpleMotionPath class defines the name of the property, the property’s starting value, and the property’s ending value.

You can define multiple instance of the SimpleMotionPath class to animate multiple properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkAnimate2Prop.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Animate id="scaleIncrease"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.0" valueTo="1.5"/>
            <s:SimpleMotionPath property="scaleY"
                valueFrom="1.0" valueTo="1.5"/>
        </s:Animate>
        <s:Animate id="scaleDecrease"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.5" valueTo="1.0"/>
            <s:SimpleMotionPath property="scaleY"
                valueFrom="1.5" valueTo="1.0"/>
        </s:Animate>
    </fx:Declarations>
    <s:Button id="myB1"
        label="Scale Button"
        mouseDown="scaleDecrease.end(); scaleIncrease.play();"
        mouseUp="scaleIncrease.end(); scaleDecrease.play();"/>
</s:Application>
```

The Fade, Resize, and AnimateColor effects

The Fade, Resize, and AnimateColor effects modify specific properties of the target. These effects are predefined to modify specific properties. So, you do not need to use the SimpleMotionPath class to specify the property, as you do for the Animate effect.

The following example uses the Resize effect on an Image control:

```
<?xml version="1.0"?>
<!-- behaviors\SparkResizeEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:Resize id="myResizeEffect"
            target="{myImage}"
            widthBy="10" heightBy="10"/>
    </fx:Declarations>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Resize Me"
        click="myResizeEffect.end();myResizeEffect.play();"/>
</s:Application>
```

This example resizes the Image control by increasing its height and width by 10 pixels when the user clicks the Button control.

The Resize effect defines six properties that you can use to configure it: `heightBy`, `widthBy`, `heightFrom`, `widthFrom`, `heightTo`, and `widthTo`. If omitted, Flex automatically calculates any values for these properties. In the previous example, `heightFrom` and `widthFrom` are set to the current size, and `heightTo` and `widthTo` are set to the final size.

Spark transform effects

The Spark property effects work by modifying one or more properties of the target over the duration of the effect. Using the property effects, you can define two effects that play in parallel but modify the same property of the target. In this situation, the two effects conflict with each other and the results of the effects are undefined.

The Spark transform effects are designed to work together. You can define multiple effects that, while playing in parallel, do not interfere with each other. When executing multiple transform effects in parallel, the effects are combined into a single transformation, rather than playing independently. By combining the transform effects, Flex eliminates the chance that one effect could interfere with another.

The Spark transform effects include the Move, Rotate, and Scale effects.

Applying transform effects

The following example applies the Rotate and Move effects in parallel to an Image control:

```
<?xml version="1.0"?>
<!-- behaviors\SparkXFormRotateMove.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Parallel id="parallelRMForward"
            target="{myImage}">
            <s:Rotate
                angleBy="180"/>
            <s:Move
                xBy="100"
                yBy="100"/>
        </s:Parallel>
        <s:Parallel id="parallelRMBack"
            target="{myImage}">
            <s:Rotate
                angleBy="180"/>
            <s:Move
                xBy="-100"
                yBy="-100"/>
        </s:Parallel>
    </fx:Declarations>
    <s:Button label="Play Effect Forward"
        x="10" y="10"
        click="parallelRMForward.end();parallelRMForward.play();"/>
    <s:Button label="Play Effect Backward"
        x="150" y="10"
        click="parallelRMBack.end();parallelRMBack.play();" />
    <s:Image id="myImage"
        x="10" y="50"
        source="@Embed(source='assets/logo.jpg')"/>
</s:Application>
```

Transform effects operate relative to the *transform center* of the target. By default, the transform center of the target is the upper-left corner of the target, corresponding to coordinates (0,0) in the target coordinate system.

For the Move effect, the change in position is measured by a change of the location of the transform center of the target. The Rotate effect rotates the target around the transform center, and the Scale effect scales the target while the transform center of the target stays stationary.

You do not have to use the upper-left corner of the target as the transform center. For example, instead of rotating the target around the upper-left corner, you rotate the target around its center point.

You can set the coordinates of the transform center either on the target itself, or on the effect class. If you set the transform center on the target, all transform effects use that transform center. Use the UIComponent properties `transformX`, `transformY`, and `transformZ` to define the transform center of the target.

If you set the transform center on the effect, the effect uses that transform center for the duration of the effect. Use the `transformX`, `transformY`, and `transformZ` properties on the transform effect classes to set the transform center for all effect targets. Setting these properties on the effect class overrides the corresponding setting on the target.

You do not have to specify all three properties. For example, if you only want to specify the x location of the transform center, set only the `transformX` property on the target or on the effect class.

You can also set the effect's `autoCenterTransform` property to `true` to configure the effect to play the transform around the center point of the target. If you apply the effect to multiple targets, Flex calculates the transform center independently for each target.

If you apply multiple transform effects in parallel to the same target, set the transform center to the same value for all the effects. For example, set the `autoCenterTransform` property to `true` for one transform effect, set it to the same value for all effects.

The next example modifies the previous example set the `autoCenterTransform` property to `true`. The effects now operate on the center of the target:

```
<?xml version="1.0"?>
<!-- behaviors\SparkXFormRotateMoveAutoCenter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Parallel id="parallelRMForward"
            target="{myImage}">
            <s:Rotate
                angleBy="180"
                autoCenterTransform="true"/>
            <s:Move
                xBy="100"
                yBy="100"
                autoCenterTransform="true"/>
        </s:Parallel>
        <s:Parallel id="parallelRMBack"
            target="{myImage}">
            <s:Rotate
                angleBy="180"
                autoCenterTransform="true"/>
            <s:Move
                xBy="-100"
                yBy="-100"
                autoCenterTransform="true"/>
        </s:Parallel>
    </fx:Declarations>
    <s:Button label="Play Effect Forward"
        x="10" y="10"
        click="parallelRMForward.end();parallelRMForward.play();"/>
    <s:Button label="Play Effect Backward"
        x="150" y="10"
        click="parallelRMBack.end();parallelRMBack.play();" />
    <s:Image id="myImage"
        x="10" y="50"
        source="@Embed(source='assets/logo.jpg')"/>
</s:Application>
```

Limitations with transform effects

The only limitation of transform effects is that they cannot be played multiple times. Therefore, you cannot use the `repeatCount`, `repeatDelay`, and `repeatBehavior` properties with transform effects.

Spark 3D effects

Most effects manipulate the effect target in the x and y dimensions to create two-dimensional effects. In the two-dimensional coordinate system, the x, y coordinates of 0, 0 corresponds to the upper-left corner of the component's coordinate system. For example, if the Application container takes up your full computer screen, those coordinates correspond to the upper-left corner of the computer screen. Increasing values of x moves to the right of the compute screen, and increasing values of y move down the screen.

The 3D effects add support for the z-axis. The z = 0 coordinate corresponds to the plane of the computer screen. Increasing values of z moves an object into the screen, making the object look farther away from the viewer. Decreasing values of z move the object toward the viewer.

The spark 3D effects are transform effects designed to take advantage of the support for three-dimensional graphics in Flash Player. Flex includes the following 3D effects:

- Move3D

  Moves the effect target in the x, y, and z coordinate system. Moving the target to increasing values in the z direction makes it appear to move back away from the viewer, so the target shrinks. Moving the target to decreasing values in the z direction makes it appear to move toward the viewer, so the target grows.

- Rotate3D

  Rotates the effect target around the x, y, or z-axis. For example, rotating the target around the y-axis rotates the object vertically through the x and z planes, similar to a door opening and closing on vertical hinges. Rotating the target around the z-axis makes the object rotate through the x and y planes, which is the same as a two-dimensional rotation.

- Scale3D

  Scales the effect target in the x, y, and z directions by setting the appropriate scale properties on the target. A scale of 2.0 means the object has been magnified by a factor of 2. A scale of 0.5 means the object has been reduced by a factor of 2. A scale value of 0.0 is not allowed.

For an introduction to transform effects, see "Spark transform effects" on page 1808.

Applying 3D effects

The following example applies the Move3D effect to an image:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DMoveEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Move3D id="moveEffect" target="{targetImg}"
            xBy="100" zBy="100"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>
    <s:Panel title="Move3D Effect Example"
        width="75%" height="75%" >

        <s:Image id="targetImg"
            horizontalCenter="0"
            verticalCenter="0"
            source="@Embed(source='assets/Nokia_6630.png')"/>

        <s:Button id="playButton"
            left="5" bottom="5"
            label="Move3D"
            click="moveEffect.play();"/>
    </s:Panel>
</s:Application>
```

In this example, the Move3D effect moves the target by increasing the value of the x and z coordinates by 100. It then reverses, and moves the component back to its original x and z coordinates. To the user, the image moves to the left and shrinks, then moves back to the right and grows to its original size.

The following example uses the Rotate3D effect to rotate the image around the y-axis:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>
    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:Image id="targetImg"
            horizontalCenter="0"
            verticalCenter="0"
            source="@Embed(source='assets/Nokia_6630.png')"/>
        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Setting the transform center of a 3D effect

By default, the transform center of the target of a 3D transform effect is the upper-left corner of the target component. This point corresponds to coordinates (0, 0, 0) in the target component's coordinate system. You can set the transform center to a different location. For an introduction to setting the transform center for two-dimensional effects, see "Applying transform effects" on page 1808.

If you run the example in the previous section, you notice that the target object rotates around its left edge. That is because the effects, by default, operates around the default transform center of the target object.

Often, you want to rotate the target component around its center point instead of around the default transform center. One option is to set the transform center to the center point of the component by setting the `autoCenterTransform` property to `true`. The following example modifies the example from the previous section to rotate the image around its center point:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateLayoutCenterTransform.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            duration="3000"
            autoCenterTransform="true"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>
    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>
        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Alternatively, you can set the `transormX`, `transformY`, and `transformZ` properties on the effect target to define the transform center. In the following example, you set the transform center of the image to the right edge, corresponding to the `transormX` property having a value of 100:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateLayoutRightTransform.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            duration="3000"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>
    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image id="targetImg"
                transformX="100"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>
        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Component layout and 3D effects

Effects can modify the layout of the parent container of the effect target. For example, suppose the effect target is in a container that uses vertical layout. You then use the two-dimensional Rotate effect to rotate the target through 360°. As the effect plays, the parent container modifies the layout of its other children to accommodate the rotating child. Therefore, container children can change position during the effect.

Flex has the concept of layering of the children of a container. Children are drawn on the screen in the order in which they are defined in the container. If children overlap, the child defined later in the container appears on top because it is drawn last.

Flex provides a set of built-in layout classes, such as the VerticalLayout class, to control child layout in a container. These layout classes assume that all children are in the z = 0 plane. That means a container always lays out children in the two-dimensional x, y plane.

However, a 3D effect can modify the effect target in the x, y, and z dimensions. If your 3D effect uses the z dimension, the built-in layout classes do not consider it during layout. If you then allow the parent container to update its layout by taking only the x and y dimensions into consideration, your application might not appear correctly. Therefore, you typically disable the parent container from performing layout while the 3D effect plays.

All effects support the `disableLayout` property. When set to `true`, this property disables layout in the parent container of the effect target for the duration of the effect. The default value is `false`. For effects though, you typically do not want to disable layout of the parent container entirely.

All transform effects define the `applyChangesPostLayout` property which, by default, is set to `true` for the 3D effects. This setting lets the 3D effect modify the target component, but the parent container ignores the changes and does not update its layout while the effect plays. Changes to other container children still cause a layout update.

Note: *For the 2D transform effects Move, Rotate, and Scale, the affectLayout property is true by default.*

You can think of the 3D effects as not playing until after the container for the effect target has completed its layout. Because the effect plays post layout, the parent container does not modify its layout for changes to the target component caused by the effect.

In the following example, the application contains three images in an HGroup container. The Rotate3D effect then plays on the middle image to rotate it around the y-axis. Because the `applyChangesPostLayout` property is `true` by default, no layout occurs as the image rotates, and the target image overlaps the image on its left:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateLayout.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            duration="3000"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>
    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>
        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

For example, set the `transformX` property to 100 on the effect target in the previous example to rotate the middle image so that it overlaps the image on the right. The image on the right appears on top of the middle image because it was drawn last.

You can override the default of disabling layout on the effect target by setting the `applyChangesPostLayout` property of the effect to `false`. In the following example, you rotate the image around the z-axis, with the `applyChangesPostLayout` property set to `false`. A 3D rotation around the z-axis is essentially a 2D rotation in the x and y plane. Because the `applyChangesPostLayout` property is `false`, the parent container updates the layout of the other two images as the effect plays:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateWithLayout.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleZFrom="0" angleZTo="360"
            applyChangesPostLayout="false"
            duration="5000"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>
    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>
        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Setting the postLayoutTransformOffsets property on a component

You can directly modify the position, rotation, and scale of a component post layout without using a 3D effect. Instead, use the `postLayoutTransformOffsets` property, of type mx.geom:TransformOffsets, of the UIComponent class. By setting the `postLayoutTransformOffsets` property directly, you modify the target without causing the parent container to update its layout.

In the following example, you use the `postLayoutTransformOffsets` property to modify the position and scale of a component. As you modify it, notice that the parent container does not update its layout:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DOffset.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.geom.TransformOffsets;

            // Define an instance of TransformOffsets.
            private var myXForm:TransformOffsets = new TransformOffsets();

            // Initialize the postLayoutTransformOffsets property of the target.
            private function initOffsets():void {
                targetImg.postLayoutTransformOffsets = myXForm;
            }

            // Move the target 20 pixels to the left and
            // increase its x and y scale by 0.1.
            private function nudgeImageLeft():void {
                targetImg.postLayoutTransformOffsets.x =
                    targetImg.postLayoutTransformOffsets.x - 20;
                targetImg.postLayoutTransformOffsets.scaleX =
                    targetImg.postLayoutTransformOffsets.scaleX + 0.1;
                targetImg.postLayoutTransformOffsets.scaleY =
                    targetImg.postLayoutTransformOffsets.scaleY + 0.1;
            }
            // Move the target 20 pixels to the right and
            // decrease its x and y scale by 0.1.
            private function nudgeImageRight():void {
                targetImg.postLayoutTransformOffsets.x =
                    targetImg.postLayoutTransformOffsets.x + 20;
                targetImg.postLayoutTransformOffsets.scaleX =
                    targetImg.postLayoutTransformOffsets.scaleX - 0.1;
                targetImg.postLayoutTransformOffsets.scaleY =
                    targetImg.postLayoutTransformOffsets.scaleY - 0.1;
            }
            // Reset the transform.
            private function resetImage():void {
                targetImg.postLayoutTransformOffsets.x = 0;
                targetImg.postLayoutTransformOffsets.scaleX = 1.0;
                targetImg.postLayoutTransformOffsets.scaleY = 1.0;
            }
        ]]>
    </fx:Script>
    <s:Panel title="Offset Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <s:Image
```

```
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <s:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"
                creationComplete="initOffsets();"/>
            <s:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>
        <s:HGroup left="5" bottom="5">
            <s:Button id="nudgeLeftButton"
                label="Nudge Left"
                click="nudgeImageLeft();"/>
             <s:Button id="nudgeRightButton"
                label="Nudge Right"
                click="nudgeImageRight();"/>
             <s:Button id="resetButton"
                label="Reset"
                click="resetImage();"/>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

Notice in this example that as you nudge the image to the left, it overlaps the image to its left. As you nudge the image to the right, it moves behind the image on the right. This overlap is because the image on the right is defined later in the container, and is therefore drawn last on the screen.

Setting the center of the projection

The 3D effects work by mapping a three-dimensional image onto a two-dimensional representation for display on a computer screen. The *projection point* defines the center of the field of view, and controls how the target is projected from three dimensions onto the screen.

By default, when you apply a 3D effect, the effect automatically sets the projection point to the center of the target. You can set the `autoCenterProjection` property of the effect to `false` to disable this default. You then use the `projectionX` and `projectionY` properties to explicitly set the projection point. These properties specify the offset of the projection point from the (0, 0) coordinate of the target.

Spark pixel-shader effects

The Spark pixel-shader effects apply an animation to a target that has a before and after bitmap representation, rather than by animating a change to the value of a property. The effect works by capturing a before bitmap image of the component, capturing an after bitmap image of the component, and then applying the animation between the two images.

While the use of pixel-shader effects might seem obvious when working with Image components, you can use them with any component.

The bitmaps are represented by an instance of the flash.display.BitmapData class. The animation from the initial to the final bitmap is defined by a pixel-shader program created by using Adobe® Pixel Bender™ Toolkit. A pixel-shader program used with the pixel-shader effects operates on two bitmap inputs to animate from one to the other.

The Spark pixel-shader effects, CrossFade and Wipe, use their own internal pixel-shaders. However, you can use a custom pixel-shader defined by using the Pixel Bender Toolkit. For an example of a custom pixel-shader, see “Creating a custom pixel shader” on page 1821.

You can use pixel-shader effects on their own, just as you can use the Spark effects such as Fade and Resize. However, since they are designed to by applied to a component with a before and after bitmap, they are often used as part of a transition.

Using a pixel-shader effect in a transition

View states let you change the appearance of an application, typically in response to a user action. Transitions define how a change of view state looks as it occurs on the screen. You define a transition by using the effect classes, in combination with several effects designed explicitly for handling transitions. For more information on view states, see "View states" on page 1847. For more information on transitions, see "Transitions" on page 1870.

When using a pixel-shader effect with a transition, you use one view state to represent the before image of the target, and another view state to represent the after image. The following example uses the Wipe effect as part of a transition that changes the font color of a button control from black to red:

```
<?xml version="1.0"?>
<!-- behaviors\SparkWipeEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <!-- Define two view states.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="red"/>
    </s:states>
    <!-- Define the transition that applies the Wipe effect
        whenever the view state changes.-->
    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence target="{myB}">
                <s:Wipe id="wipeEffect"
                    direction="right"
                    duration="1000"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>
    <s:Button id="myB" label="Wipe"
        color="black" color.red="red"/>
    <!-- Define two buttons to change the view state. -->
    <s:Button label="Set Default State"
        click="currentState='default'"/>
    <s:Button label="Set Red State"
        click="currentState='red'"/>
</s:Application>
```

The transition automatically captures the bitmap of the button in the initial and final states, then plays the Wipe effect on the button to animate the change.

Applying a pixel-shader effect

To use a pixel-shader effect outside a transition, create the bitmaps that define the initial state of the target and the final state of the target, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkShaderBitmap.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

        import mx.core.BitmapAsset;

        [Embed(source='assets/logo.jpg')]
        [Bindable]
        public var beforeImage:Class;
        [Bindable]
        public var beforeImageBitmap:BitmapAsset = new beforeImage();
        [Embed(source='assets/flexLogo.jpg')]
        [Bindable]
        public var afterImage:Class;
        [Bindable]
        public var afterImageBitmap:BitmapAsset = new afterImage();

        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:CrossFade id="forwardCF"
            target="{myImage}"
            bitmapFrom="{beforeImageBitmap.bitmapData}"
            bitmapTo="{afterImageBitmap.bitmapData}"
            effectEnd="myImage.source=afterImage;"/>
        <s:CrossFade id="backwardCF"
            target="{myImage}"
            bitmapFrom="{afterImageBitmap.bitmapData}"
            bitmapTo="{beforeImageBitmap.bitmapData}"
            effectEnd="myImage.source=beforeImage;"/>
    </fx:Declarations>
    <s:Image id="myImage"
        source="{beforeImage}"/>
    <s:Button id="fwd" label="Forward"
        click="forwardCF.end();forwardCF.play();"/>
    <s:Button id="bwd" label="Backward"
        click="backwardCF.end();backwardCF.play();"/>
</s:Application>
```

In this example, you embed two JPEG images, logo.jpg and flexLogo.jpg. When you embed JPEG files into Flex, they are represented as instances of the mx.core.BitmapAsset class. Use the `bitmapData` property of the BitmapAsset class to access the BitmapData object that contains the actual image data. For more information on embedding JPEG files, see "Embedding JPEG, GIF, and PNG images" on page 1707.

Note: *As written, this example causes the compiler to issue warnings for the binding to the* `bitmapData` *property. To remove these warnings, compile the application with the* `show-binding-warnings` *compiler flag set to* `false`*. Or, create a custom object to represent the image. For more information, see "Using data binding with Objects" on page 309.*

The first CrossFade effect, forwardCF, uses logo.jpg file to define the before state and the flexLogo.jpg to define the final state of the effect. When it plays, the effect animates the transition from the logo.jpg file to the linelogo.jpg file. After the effect plays, it loads flexLogo.jpg into the Image control. The backwardCF effect does the opposite.

The target of the effect is the Image control. When the effect plays, the following occurs:

1 The effect target, the Image control, is hidden.

2 The bitmap for the initial state displays.

3 The pixel-shader plays to animate the change from the initial state to the final state.

4 When the effect ends, hide the bitmap for the final state and make the Image control visible.

Creating a custom pixel shader

You can use the Pixel Bender Toolkit to create a custom pixel shader, and then pass it to the AnimateTransitionShader effect, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkCustomPBTransform.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

     <fx:Script>
          <![CDATA[
            // Embed the .pbj file.
            [Embed(source="assets/twist.pbj", mimeType="application/octet-stream")]
            private static var CustomShaderClass:Class;
            [Bindable]
            private static var customShaderCode:ByteArray = new CustomShaderClass();
          ]]>
     </fx:Script>

     <fx:Declarations>
        <!-- Use the custom pixel shader with an effect. -->
        <s:AnimateTransitionShader  id="shadeAnim"
            shaderByteCode="{customShaderCode}"
            target="{btn2}"
            repeatCount="2"
            repeatBehavior="reverse"/>
     </fx:Declarations>
     <s:Button id="btn2" label="Click Me" click="shadeAnim.play();"/>
</s:Application>
```

A custom pixel shader is represented by a .pbj file created by using the Pixel Bender Toolkit. To use the .pbj file in your application, you embed it, and then assign it to a property of type ByteArray. You then specify the ByteArray object as the value of the `shaderByteCode` property of the AnimateTransitionShader effect.

The AnimateTransitionShader effect also supports the `shaderProperties` property that lets you pass properties directly to the pixel shader.

The .pbj file has the following requirements:

- Define three `image4` inputs. The first input is unused but must be defined and referenced in the code. If the input is not referenced, the compiler in the Pixel Bender Toolkit removes it as part of optimizing the code.

  One `image4` input must be called `from`, and one named `to`. These inputs represent the initial and final states of the target.

- Define one parameter named `progress` of type float. This parameter represents the current fraction of the animation completed from 0.0, animating started, to 1.0, animation completed.

The following example shows the source .pbk file for the twist.pbj file used in the previous example:

```
<languageVersion : 1.0;>
kernel FxTwist
<   namespace : "flex";
    vendor : "Adobe";
    version : 1;
    description : "Twisty Effect";
>
{
    input image4 src0;
    input image4 from;
    input image4 to;
    output pixel4 dst;

    parameter float progress<
        minValue: 0.00;
        maxValue: 1.00;
        defaultValue: 0.0;
    >;

    parameter float width<
        minValue: 0.0;
        maxValue: 1024.0;
        defaultValue: 180.0;
    >;

    parameter float height<
        minValue: 0.00;
        maxValue: 1024.0;
        defaultValue: 275.0;
    >;

    void
    evaluatePixel()
    {

        // Common initialization
        float2 outCoord = outCoord();
        pixel4 color1 = sampleNearest(src0, outCoord);
        const float _height = 2.0;
        const float scale = 1.0 + _height;
        float start = progress * scale - _height;
        float end   = start + _height;


        float yfrac = outCoord.y / height;
        float angle = (yfrac - start) / (end - start);
        float _width = cos(angle * 3.141592653589);

        if (yfrac < start) {
            dst = sampleLinear(to, outCoord());
        }
        else if (yfrac > end) {
            dst = sampleLinear(from, outCoord());
        }
        else {
        if (_width > 0.0) {

            float xstep  = (65536.0 / _width);
            float xval   = 0.0;
            float startx = (width - width * _width) / 2.0;
            float xcount = startx;

            float sampleWidth = (width * 65536.0) / (xstep);

            if (outCoord.x < startx)
                dst = float4(0.0,0.0,0.0,0.0);
            else if (outCoord.x < (startx + sampleWidth)) {
                float perc = (outCoord.x - startx) / sampleWidth;
                dst = sampleLinear(to, float2(width * perc, outCoord.y));
            }
            else
                dst = float4(0.0,0.0,0.0,0.0);
        }

        else if (_width < 0.0) {
            float xstep  = (65536.0 / -_width);
            float xval   = 0.0;
            float startx = (width + width * _width) / 2.0;
            float xcount = startx;
            float sampleWidth = (width * 65536.0) / (xstep);

            if (outCoord.x < startx)
                dst = float4(0.0,0.0,0.0,0.0);
            else if (outCoord.x < (startx + sampleWidth)) {
                float perc = (outCoord.x - startx) / sampleWidth;
                dst = sampleLinear(from, float2( width * perc, outCoord.y));
            }
            else
                dst = float4(0.0,0.0,0.0,0.0);
        }

        }
    }
}
```

You can also look at the source code for the CrossFade.pbk file in the frameworks\projects\spark\src\spark\effects directory of the Flex source code.

Spark filter effects

Filters let you modify the visual appearance of your components not by modifying properties of the component but by applying a visual filter to the component. The filters are defined in the spark.filters. package. Common filters include the DropShadowFilter, GlowFilter, BlurFilter, and the other filter classes in the spark.filters. package.

Filters define a static change to the component where the properties of the filter are fixed unless you explicitly modify them at runtime. You can use the AnimateFilter effect to animate a filter at runtime. Unlike effects that animate properties of the target, the AnimateFilter effect animates properties of the filter applied to the target.

For example, if you apply the DropShadowFilter to a component, it has a static value for the filter properties such as `color`, `distance`, and `angle`. If you use the AnimateFilter effect to apply the filter to the target, you can animate these properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkAnimateDropShadowFilter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:DropShadowFilter id="myDSF"
            color="0x0000FF"
            distance="5"
            angle="315"/>
        <s:AnimateFilter id="myFilter"
            target="{myB2}"
            repeatCount="0"
            duration="500"
            repeatBehavior="reverse"
            bitmapFilter="{new spark.filters.DropShadowFilter()}">
            <s:SimpleMotionPath property="color" valueFrom="0" valueTo="0x0000FF"/>
            <s:SimpleMotionPath property="distance" valueFrom="0" valueTo="10"/>
            <s:SimpleMotionPath property="angle" valueFrom="270" valueTo="360"/>
        </s:AnimateFilter>
    </fx:Declarations>

    <s:Button id="myB1"
        x="50" y="50"
        label="Show a DropShadowFilter"
        filters="{[myDSF]}"/>
    <s:Button id="myB2"
        x="50" y="95"
        label="Animate a DropShadowFilter"
        click="myFilter.end();myFilter.play();"/>
</s:Application>
```

In this example, the first button uses a static DropShadowFilter. The second button uses the AnimateFilter effect to animate the filter. Because the effect sets the `repeatCount` to 0, the effect plays continuously.

To use the AnimateFilter effect, you specify the filter to animate, and then use the SimpleMotionPath class to specify the properties of the filter to animate. In this example, the effect animate the `color`, `distance`, and `angle` properties.

Motion paths and keyframes

The Spark effect classes are designed to make it easy to specify the property to animate, and the starting and ending values of the property for the animation. These classes hide much of the underlying effects implementation to simplify the use of effects.

However, you might want to create an effect that animates a property over a set of values, rather than over just a starting and ending value. To create these effect, use keyframes and motion paths.

A *keyframe* defines the value of a property at a specific time during the effect. For example, you can create three keyframes that define the value of a property at the beginning of the effect, at the midpoint of the effect, and at the end of the effect. The effect animates the property change on the target from keyframe to keyframe over the effect duration.

If your effect has just two keyframes, use the Animate effect. The Animate effect takes a starting an ending value for the property, corresponding to two keyframes. For more information, see “The Animate effect” on page 1806.

The collection of keyframes for an effect is called the effect’s motion path. A *motion path* can define any number of keyframes. The effect then calculates the value of the property by interpolating between the values specified by two keyframes.

Applying an effect using keyframes and motion paths

Use the Keyframe class to define a keyframe. You typically define a keyframe by specifying the value of the property, and the time during the duration of the effect when the property has that value.

The MotionPath class contains a Vector of Keyframe objects. The MotionPath class also defines the name of the property that is modified by the effect, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkKeyFrame.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <fx:Vector id="kf" type="spark.effects.animation.MotionPath">
            <s:MotionPath property="scaleX">
                <s:Keyframe time="250" value="0.5"/>
                <s:Keyframe time="500" value="1.0"/>
                <s:Keyframe time="750" value="0.5"/>
                <s:Keyframe time="1000" value="1.0"/>
                <s:Keyframe time="1250" value="0.5"/>
                <s:Keyframe time="1500" value="1.0"/>
            </s:MotionPath>
            <s:MotionPath property="scaleY">
                <s:Keyframe time="250" value="0.5"/>
                <s:Keyframe time="500" value="1.0"/>
                <s:Keyframe time="750" value="0.5"/>
                <s:Keyframe time="1000" value="1.0"/>
                <s:Keyframe time="1250" value="0.5"/>
                <s:Keyframe time="1500" value="1.0"/>
            </s:MotionPath>
        </fx:Vector>

        <s:Animate id="shrinkEffect"
            motionPaths="{kf}"
            target="{myImage}"/>
    </fx:Declarations>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"
        click="shrinkEffect.end();shrinkEffect.play();"/>
</s:Application>
```

Each MotionPath object defines the property of the target to animate, and contains six Keyframe objects. The Keyframe objects define the values of the `scaleX` and `scaleY` properties of the target at different times during the effect. The Animate class takes as the value of the `motionPaths` property a Vector of MotionPath objects.

In most situations, you do not have to work with keyframes and motion paths; the Spark effect classes handles the creating and use of keyframes internally. For example, the Animate class can take as the value of the `motionPaths` property a Vector of SimpleMotionPath objects. Each SimpleMotionPath object defines the property to animate, the starting property value, and the ending property value.

The SimpleMotionPath class is a subclass of the MotionPath class. When you create an instance of the SimpleMotionPath class, it creates two instances of the Keyframe class to represent the beginning and ending values of the property on the target. The effect then animates the property change on the target between the values specified by the two Keyframe objects.

Handling Spark effect events

Spark effects support all the general events, such as `effectStart`, `effectStop`, and `effectEnd`. For more information on handling these events, see "Handling effect events" on page 1796.

In addition, the Spark effect classes support the following additional events:

- `effectRepeat` For any effect that is repeated more than once, dispatched when the effect begins a new repetition. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_UPDATE`.
- `effectUpdate` Dispatched every time the effect updates the target. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_UPDATE`.

Using Spark easing classes

You can change the acceleration of an effect animation by using an easing class with an effect. With easing, you can create a more realistic rate of acceleration and deceleration. You can also use an easing class to create a bounce effect or control other types of motion.

Flex supplies the Spark easing classes in the spark.effects.easing package. This package includes classes for the most common types of easing, including Bounce, Linear, and Sine easing. For more information on using these classes, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Easing class	Description
Bounce	The movement of the effect target accelerates toward the end value, and then bounces against the end value several times.
Elastic	The effect target decelerates toward the end value, and continues past the end value. It then oscillates around the end value in smaller and smaller increments, before reaching the end value.
Linear	Defines three phases of animation: acceleration, uniform motion, and deceleration. Acceleration and deceleration occur at a constant rate. Use the `easeinFraction` property to specify the percentage of the total animation duration for acceleration. Use the `easeOutFraction` property to specify the percentage of the total animation duration for deceleration.
Power	Defines easing as consisting of two phases: the acceleration, or ease in phase, followed by the deceleration, or ease out phase. The rate of acceleration and deceleration is based on the `exponent` property. The higher the value of the `exponent` property, the greater the acceleration and deceleration. Use the `easeInFraction` property to specify the percentage of an animation accelerating.
Sine	Defines easing as consisting of two phases: the acceleration, or ease in phase, followed by the deceleration, or ease out phase. Use the `easeInFraction` property to specify the percentage of an animation accelerating.

The following example uses the Sine and Power easing classes with the Move effect:

```
<?xml version="1.0"?>
<!-- behaviorExamples\SparkResizeEasing.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="750">
    <fx:Declarations>
        <s:Sine id="sineEasing"
            easeInFraction="0.3"/>
        <s:Power id="powerEasing"
            exponent="4"/>
        <s:Move id="moveRight"
            target="{myImage}"
            xBy="500"
            duration="2000"
            easer="{powerEasing}"/>
        <s:Move id="moveLeft"
            target="{myImage}"
            xBy="-500"
            duration="2000"
            easer="{sineEasing}"/>
    </fx:Declarations>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Move Right"
         x="0" y="100"
        click="moveRight.end();moveRight.play();"/>
    <s:Button label="Move Left"
         x="0" y="125"
        click="moveLeft.end();moveLeft.play();"/>
</s:Application>
```

The Sine effect specifies a short acceleration phase for the move effect, followed by a longer deceleration. The Power effect specifies a rapid acceleration for the move.

The following example uses the Bounce and Elastic easing classes:

```
<?xml version="1.0"?>
<!-- behaviorExamples\SparkBounceEasing.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="750">
    <fx:Declarations>
        <s:Bounce id="bounceEasing"/>
        <s:Elastic id="elasticEasing"/>
        <s:Move id="moveRight"
            target="{myImage}"
            xBy="500"
            duration="2000"
            easer="{elasticEasing}"/>
        <s:Move id="moveLeft"
            target="{myImage}"
            xBy="-500"
            duration="2000"
            easer="{bounceEasing}"/>
    </fx:Declarations>
    <s:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Move Right"
         x="0" y="100"
        click="moveRight.end();moveRight.play();"/>
    <s:Button label="Move Left"
         x="0" y="125"
        click="moveLeft.end();moveLeft.play();"/>
</s:Application>
```

Former Adobe engineer Chet Haase describes custom easing in Custom Easing in Flex 4 Effects and using transitions with easing classes in Transitions and Easing in Flex.

Using MX effects

In MX, an effect defines the visual or audible change to the target component, and a trigger initiates the effect. A trigger is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. The MX effects are shipped in the mx.effects package.

While Flex ships with both Spark and MX effects, Adobe recommends that you use the Spark effects when possible.

For an introduction to effects, see “Introduction to effects” on page 1784.

Available MX triggers

You use a trigger name to assign an effect to a target component. You can reference a trigger name as a property of an MXML tag, in the `<fx:Style>` tag, or in an ActionScript `setStyle()` and `getStyle()` function. Trigger names use the following naming convention:

triggerEvent`Effect`

where *triggerEvent* is the event that invokes the effect.

For example, the `focusIn` event occurs when a component gains focus; you use the `focusInEffect` trigger property to specify the effect to invoke for the `focusIn` event. The `focusOut` event occurs when a component loses focus; the corresponding trigger property is `focusOutEffect`.

The following table lists the effect name that corresponds to each trigger:

Trigger name	Triggering event
`addedEffect`	Component is added as a child to a container.
`creationCompleteEffect`	Component is created.
`focusInEffect`	Component gains keyboard focus.
`focusOutEffect`	Component loses keyboard focus.
`hideEffect`	Component becomes invisible by changing the `visible` property of the component from `true` to `false`.
`mouseDownEffect`	User presses the mouse button while the mouse pointer is over the component.
`mouseUpEffect`	User releases the mouse button.
`moveEffect`	Component is moved.
`removedEffect`	Component is removed from a container.
`resizeEffect`	Component is resized.
`rollOutEffect`	User rolls the mouse pointer off the component.
`rollOverEffect`	User rolls the mouse pointer over the component.
`showEffect`	Component becomes visible by changing the `visible` property of the component from `false` to `true`.

Applying MX effects in MXML

Associate MX effects with triggers as part of defining the basic behavior for your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWL.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define effect. -->
        <mx:WipeLeft id="myWL" duration="1000"/>
    </fx:Declarations>
    <!-- Assign effect to targets. -->
    <mx:Button id="myButton" label="Click Me"
        mouseDownEffect="{myWL}"/>
    <mx:Button id="myOtherButton" label="Click Me"
        mouseDownEffect="{myWL}"/>
</s:Application>
```

In this example, the effect is a WipeLeft effect with a duration of 1000 milliseconds (ms). That means it takes 1000 ms for the effect to play from start to finish.

You use data binding to assign the effect to the `mouseDownEffect` property of each Button control. The `mouseDownEffect` property is the effect trigger that specifies to play the effect when the user clicks the control using the mouse pointer. In the previous example, the effect makes the Button control appear as if it is being wiped onto the screen from right to left.

Using ActionScript, you can create, modify, or play an effect. With ActionScript, you can configure the effect to play in response to an effect trigger, or you can explicitly invoke it by calling the `play()` method of the effect's class. ActionScript gives you control of effects so that you can configure them as part of a user preferences setting, or modify them based on user actions. The following example creates the WipeLeft effect in ActionScript:

```
<?xml version="1.0"?>
<!-- behaviors\AsEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="createEffect(event);" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Define effect. -->
    <fx:Script>
        <![CDATA[
            // Import the effect class.
            import mx.effects.*;

            // Define effect variable.
            private var myWL:WipeLeft;
            private function createEffect(eventObj:Event):void {
                // Create the WipeLeft effect object.
                myWL=new WipeLeft();

                // Set the effect duration.
                myWL.duration=1000;

                // Assign the effects to the targets.
                myButton.setStyle('mouseDownEffect', myWL);
                myOtherButton.setStyle('mouseDownEffect', myWL);
            }
        ]]>
    </fx:Script>
    <mx:Button id="myButton" label="Click Me"/>
    <mx:Button id="myOtherButton" label="Click Me"/>
</s:Application>
```

This example still uses an event to invoke the effect. To play an effect programmatically, you call the effect's `play()` method. For information on using ActionScript to configure and invoke effects, and for more information on using MXML, see "Applying behaviors in MXML" on page 130.

Applying MX effects in MXML using styles

All MXML properties corresponding to effect triggers are implemented as CSS styles. Therefore, you can also apply an effect using the `<fx:Style>` tag. For example, to set the `mouseDownEffect` property for all TextArea controls in an application, you can use a CSS type selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\MxmlTypeSel.mxml-->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|TextArea {mouseDownEffect: slowWipe;}
    </fx:Style>
    <fx:Declarations>
        <mx:WipeLeft id="slowWipe" duration="5000"/>
    </fx:Declarations>
    <mx:TextArea id="myTA"
        text="This TextArea slowly wipes in on mouseDown."/>
    <mx:TextArea id="myTA2"
        text="This TextArea control has no effect."
        mouseDownEffect="none"/>
</s:Application>
```

Setting the `mouseDownEffect` property in a component tag overrides any settings that you make in an `<fx:Style>` tag. If you want to remove the associated effect defined in a type selector, you can explicitly set the value of any trigger to `none`, as the following example shows:

```
<mx:TextArea id="myTA" mouseDownEffect="none"/>
```

You can also use a class selector to apply effects, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWLClassSel.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!-- Define a class selector for a TextArea control -->
    <fx:Style>
        .textAreaStyle { mouseDownEffect: WipeLeft; }
    </fx:Style>
    <fx:Declarations>
        <mx:WipeLeft id="slowWipe" duration="5000"/>
    </fx:Declarations>
    <mx:TextArea id="myTA"
        styleName="textAreaStyle"
        text="This TextArea control quickly wipes in."/>
</s:Application>
```

Using setStyle() and getStyle() with MX effects defined in MXML

Trigger properties are implemented as styles; therefore, you can use the setStyle() and getStyle() methods to manipulate triggers and their associated effects. The `setStyle()` method has the following signature:

```
setStyle("trigger_name", effect)
```

trigger_name String indicating the name of the trigger property; for example, `mouseDownEffect` or `focusInEffect`.

effect The effect associated with the trigger. The data type of `effect` is an Effect object, or an object of a subclass of the Effect class.

The `getStyle()` method has the following signature:

```
getStyle("trigger_name"):return_type
```

trigger_name String indicating the name of the trigger property.

return_type An Effect object, or an object of a subclass of the Effect class.

The following scenarios show how to use `getStyle()` with effects defined in MXML:

When you use MXML tag properties or the `<fx:Style>` tag to apply effects to a target, `getStyle()` returns an Effect object. The type of the object depends on the type of the effect that you specified. In the following example, you use the `setStyle()` method to set the duration of an effect, and the `getStyle()` method to return the duration:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWLGetStyleMXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            private function displayStyle():void {
                var s:String = String(myB.getStyle('mouseDownEffect').duration)
                myTA.text = "mouseDownEffect duration: " + s;
            }

            private function changeStyle(n:Number):void {
                myB.getStyle('mouseDownEffect').duration = n;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:WipeLeft id="slowWipe" duration="5000"/>
    </fx:Declarations>
    <!-- Set the behavior in MXML. -->
    <mx:Button id="myB" label="Click Me"
        mouseDownEffect="{slowWipe}"/>
    <mx:TextArea id="myTA" width="200"/>

    <!-- Call getStyle() to return an object of type WipeLeft. -->
    <mx:HBox>
        <mx:Button label="Get Style" click="displayStyle();"/>
        <mx:Button label="Set Duration" click="changeStyle(1000);"/>
        <mx:Button label="Reset Duration" click="changeStyle(5000);"/>
    </mx:HBox>
</s:Application>
```

For more information on working with styles, see “Styles and themes” on page 1492.

Applying MX effects in ActionScript using styles

Because Flex implements the properties corresponding to effect triggers as styles, you can use style sheets and the setStyle() and getStyle() methods to apply effects. Therefore, you can create an effect in ActionScript, and then use the `setStyle()` method to associate it with a trigger, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\BehaviorsASStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.effects.Zoom;
            // Define a new Zoom effect.
            private var zEffect:Zoom = new Zoom();
            private function initApp():void {
                // Set duration of zoom effect.
                zEffect.duration = 1000;
                // Define zoom in ratio.
                zEffect.zoomHeightTo = 1.0;
                zEffect.zoomWidthTo = 1.0;
            }

            private function applyZoomEffect(newZoom:Number):void {
                zEffect.zoomHeightTo = newZoom;
                zEffect.zoomWidthTo = newZoom;
                // Apply or re-apply the Zoom effect to the Button control.
                b1.setStyle("mouseDownEffect", zEffect);
            }

            private function resizeButton():void {
                var newZoom:Number;
                var n:Number = zEffect.zoomHeightTo;
                if (n == 1.0) {
                    newZoom = 2.0;
                } else {
                    newZoom = 1.0;
                }
                applyZoomEffect(newZoom);
            }
        ]]>
    </fx:Script>
    <mx:HBox>
        <mx:Button id="b1" label="Click Me" click="resizeButton();"/>
    </mx:HBox>
</s:Application>
```

You can also define the effect in MXML, then use ActionScript to apply it, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ASStylesMXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initializeEffect(event);">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;

            private function initializeEffect(eventObj:Event):void {
                myB.setStyle("mouseDownEffect", myWL);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:WipeLeft id="myWL" duration="1000"/>
    </fx:Declarations>
    <mx:Button id="myB" label="Click Me"/>
</s:Application>
```

The code in the following example alternates the WipeRight and WipeLeft effects for the `mouseDownEffect` style of a Button control:

```
<?xml version="1.0"?>
<!-- behaviors\ASStyleGetStyleMXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function changeEffect():void {
                if (myButton.getStyle("mouseUpEffect") == myWR) {
                    myButton.setStyle("mouseUpEffect", myWL);
                }
                else if (myButton.getStyle("mouseUpEffect") == myWL) {
                    myButton.setStyle("mouseUpEffect", myWR);
                }
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:WipeRight id="myWR" duration="1000"/>
        <mx:WipeLeft id="myWL" duration="1000"/>
    </fx:Declarations>
    <mx:Button id="myButton"
        label="Click Me"
        click="changeEffect();"
        mouseUpEffect="{myWL}"/>
</s:Application>
```

Handling MX effect events

MX effects support all of the general events, such as `effectStart`, `effectStop`, and `effectEnd`. For more information on handling these events, see “Handling effect events” on page 1796.

In addition, every MX effect that is a subclass of the TweenEffect class, such as the Fade and Move effects, supports the following events:

- tweenStart   Dispatched when the tween effect starts. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_START`. The `Effect.effectStart` event is dispatched before the `tweenStart` event.
- tweenUpdate   Dispatched every time a TweenEffect class calculates a new value. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_UPDATE`.
- tweenEnd   Dispatched when the tween effect ends. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_END`.

The event object passed to the event listener for these events is of type TweenEvent. The TweenEvent class is a subclass of the Event class, and contains all of the properties inherited from Event, including `target`, and `type`, and defines the following new property:

value Contains the tween value calculated by the effect. For example, for the Fade effect, the `value` property contains a single Number between the values of the `Fade.alphaFrom` and `Fade.alphaTo` properties. For the Move effect, the `value` property contains a two item Array, where the first value is the current `x` value of the effect target and the second value is the current `y` value of the effect target. For more information on the `value` property, see the instance class for each effect that is a subclass of the TweenEffect class.

Using the MX AnimateProperty effect

You use the AnimateProperty effect to animate a numeric property of a component. For example, you can use this effect to animate the `scaleX` property of a control, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\AnimateHScrollPos.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:Sequence id="animateScaleXUpDown" >
            <mx:AnimateProperty
                property="scaleX"
                fromValue="1.0"
                toValue="1.5"/>
            <mx:AnimateProperty
                property="scaleX"
                fromValue="1.5"
                toValue="1.0"/>
        </mx:Sequence>
    </fx:Declarations>
    <mx:Button label="Scale Button"
        mouseDownEffect="{animateScaleXUpDown}"/>
</s:Application>
```

In this example, clicking on the Button control starts the Sequence effect, which is made up of two AnimateProperty effects. The first AnimateProperty effect scales the control to 150% of its width, and the second scrolls it back to its original width.

Using the MX sound effect

You use the SoundEffect class to play a sound represented as an MP3 file. You specify the MP3 file using the `source` property. If you have already embedded the MP3 file, using the `Embed` keyword, you can pass the Class object of the MP3 file to the source property. Otherwise, specify the full URL to the MP3 file.

The following example shows both methods of specifying the MP3 file:

```
<?xml version="1.0"?>
<!-- behaviors\Sound.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Embed MP3 file.
            [Bindable]
            [Embed(source="../assets/sound1.mp3")]
            public var soundClass:Class;
        ]]>
    </fx:Script>

    <fx:Declarations>
        <mx:SoundEffect id="soundEmbed"
            useDuration="false"
            loops="0"
            source="{soundClass}"/>
    </fx:Declarations>
    <mx:Button id="myButton2"
        label="Sound Embed"
        mouseDownEffect="{soundEmbed}"/>
</s:Application>
```

In this example, you embed the sound1.mp3 file in your application. That means the file is compiled into the SWF file.

The SoundEffect class has several properties that you can use to control the playback of the MP3 file, including `useDuration` and `loops`. The `useDuration` property specifies whether to use the `duration` property to control the play time of the MP3 file. If the `useDuration` property is `true`, the MP3 file will play for as long as the time specified by the `duration` property, which defaults to 500 ms. If you set `useDuration` to `false`, the MP3 file plays to completion.

The `loops` property specifies the number of times to repeat the MP3 file, where a value of 0 means play the effect once, a value of 1 means play the effect twice, and so on. If you repeat the MP3 file, it still uses the setting of the `useDuration` property to determine the playback time.

The `duration` property takes precedence over the `loops` property. If the effect duration is not long enough to play the sound at least once, then the sound will not loop.

The SoundEffect class also defines the following events:

complete Dispatched when the sound file completes loading.

id3 Dispatched when ID3 data is available for an MP3 sound file.

ioError Dispatched when an error occurs during the loading of the sound file.

progress Dispatched periodically as the sound file loads. Within the event object, you can access the number of bytes currently loaded and the total number of bytes to load. The event is not guaranteed to be dispatched, which means that the `complete` event might be dispatched without any `progress` events being dispatched.

For more information, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Using MX mask effects

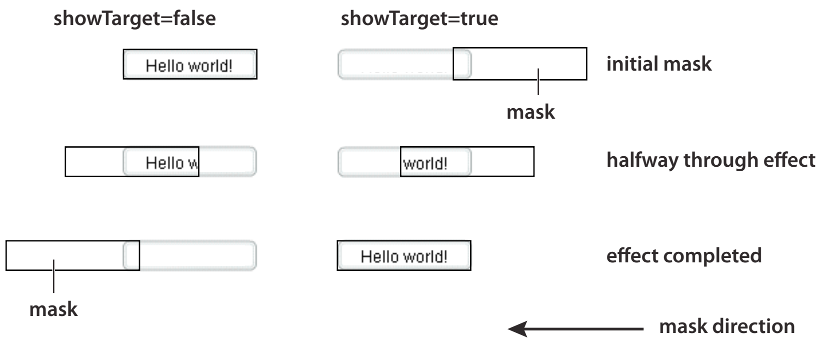
A mask effect is any effect that is a subclass of the MaskEffect class, which includes the wipe effects and the Iris effect. A mask effect uses an overlay, called a mask, to perform the effect. By default, for the wipe effects, the mask is a rectangle with the same size as the target component. For the Iris effect, the default mask is a rectangle centered on the component.

The before or after state of the target component of a mask effect must be invisible. That means a mask effect always makes a target component appear on the screen, or disappear from the screen.

To control the mask effect, you set the `MaskEffect.showTarget` property to correspond to the action of the component. If the target component is becoming visible, set `showTarget` to `true`. If the target is becoming invisible, set `showTarget` to `false`. The default value is `true`.

Often, you use these effects with the `showEffect` and `hideEffect` triggers. The `showEffect` trigger occurs when a component becomes visible by changing its `visible` property from `false` to `true`. The `hideEffect` trigger occurs when the component becomes invisible by changing its `visible` property from `true` to `false`. When using a mask effect with the `showEffect` or `hideEffect` triggers, you can ignore the `showTarget` property; Flex sets it for you automatically.

As the mask effect executes, the effect either covers the target component or uncovers it, based on the setting of the `showTarget` property. The following diagram shows the action of the WipeLeft effect for the two different settings of the `showTarget` property:



You can use several properties of the MaskEffect class to control the location and size of the mask, including the following:

scaleXFrom, scaleYFrom, scaleXTo, and scaleX Specify the initial and final scale of the mask where a value of 1.0 corresponds to scaling the mask to the size of the target component, 2.0 scales the mask to twice the size of the component, 0.5 scales the mask to half the size of the component, and so on. To use any one of these properties, you must specify all four.

xFrom, yFrom, xTo, and yTo Specify the coordinates of the initial position and final position of the mask relative to the target component, where (0, 0) corresponds to the upper-left corner of the target. To use any one of these properties, you must specify all four.

The coordinates of the initial and final position of the mask depend on the type of effect and whether the `showTarget` property is `true` or `false`. For example, for the WipeLeft effect with a `showTarget` value of `false`, the coordinates of the initial mask position are (0, 0), corresponding to the upper-left corner of the target, and the coordinates of the final position are the upper-right corner of the target (`-width`, 0), where `width` is the width of the target.

For a `showTarget` value of `true` for the WipeLeft effect, the coordinates of the initial mask position are (`width`, 0), and the coordinates of the final position are (0, 0).

Creating a custom mask function

You can supply a custom mask function to a mask effect using the createMaskFunction property. A custom mask function lets you create a mask with a custom shape, color, or other attributes for your application requirements.

The custom mask function has the following signature:

```
public function funcName(targ:Object, bounds:Rectangle):Shape
    var myMask:Shape = new Shape();
    // Create mask.

    return myMask;
}
```

Your custom mask function takes an argument that corresponds to the target component of the effect, and a second argument that defines the dimensions of the target so that you can correctly size the mask. The function returns a single Shape object that defines the mask.

The following example uses a custom mask with a WipeLeft effect:

```
<?xml version="1.0"?>
<!-- behaviors\CustomMaskSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Import the effect class.
            import mx.effects.*;

            public function createLargeMask(targ:Object,
                    bounds:Rectangle):Shape {
                // Create the Shape object.
                var largeMask:Shape = new Shape();
                // Access the Graphics object of the
                // Shape object to draw the mask.
                largeMask.graphics.beginFill(0x00FFFF, 0.5);
                largeMask.graphics.drawRoundRect(0, 0, bounds.width + 10,
                    bounds.height - 10, 3);
                largeMask.graphics.endFill();
                // Return the mask.
                return largeMask;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:WipeLeft id="customWL"
            createMaskFunction="createLargeMask"
            showTarget="false"/>

        <mx:WipeLeft id="standardWL"
            showTarget="false"/>
    </fx:Declarations>
    <mx:HBox borderStyle="solid"
        paddingLeft="10" paddingRight="10"
        paddingTop="10" paddingBottom="10">
        <mx:Button label="Custom Mask"
            mouseDownEffect="{customWL}"
            height="100" width="150"/>
    </mx:HBox>
</s:Application>
```

Using MX data effects

The MX List and MX TileList controls take input from a *data provider*, an object that contains the data displayed by the control. To provide this data, you assign a collection, which is usually an ArrayCollection or XMLListCollection object, to the control's `dataProvider` property. Each item in the control is then displayed by using an item renderer.

MX data effects make it possible to apply effects to the item renderers in List and TileList controls when the data provider for the control changes. For example, when an item is deleted from the data provider of a List control, the item renderer for that item might fade out and shrink.

For more information about data providers and controls that use data providers, see "Data providers and collections" on page 898. For more information about item renderers, see "MX item renderers and item editors" on page 1006.

By default, the List and TileList control do not use a data effect. To specify the effect to apply to the control, use the control's `itemsChangeEffect` style property. For the List control, use the DefaultListEffect class to configure the data effect. For the TileList control, use the DefaultTileListEffect class.

You can also create custom data effects. For more information, see "Custom effects" on page 2525.

Example: Applying a MX data effect to a MX List or MX TileList control

The following example applies the MX DefaultListEffect effect to the MX List control when items are added to or removed from the control. When an item in the List control is removed, this effect first fades out the item, then collapses the size of the item to 0. When you add an item to the List control, this effect expands the slot for the item, then fades in the new item.

Because the DefaultListEffect effect grows and shrinks item renderers as it plays, you must set the `List.variableRowHeight` property to `true` to enable the List control to dynamically change its row height, as the following example shows:

```
<?xml version="1.0"?>
<!-- dataEffects\ListEffectCustomDefaultEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.effects.DefaultListEffect;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection(
                ['A','B','C','D','E']);

            private function deleteItem():void {
                // As each item is removed, the index of the other items changes.
                // So first get the items to delete, and then determine their indices
                // as you remove them.
                var toRemove:Array = [];
                for (var i:int = 0; i < list0.selectedItems.length; i++)
                    toRemove.push(list0.selectedItems[i]);
                for (i = 0; i < toRemove.length; i++)
                    myDP.removeItemAt(myDP.getItemIndex(toRemove[i]));
            }
            private var zcount:int = 0;
            private function addItem():void {
                // Always add the new item after the third item,
                // or after the last item if the length is less than 3.
                myDP.addItemAt("Z"+zcount++,Math.min(3,myDP.length));
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define an instance of the DefaultListEffect effect,
             and set its fadeOutDuration and color properties. -->
        <mx:DefaultListEffect id="myDLE"
            fadeOutDuration="1000"/>
    </fx:Declarations>
    <mx:List id="list0"
        width="150"
        dataProvider="{myDP}"
        variableRowHeight="true"
        fontSize="24"
        allowMultipleSelection="true"
        itemsChangeEffect="{myDLE}"/>
    <mx:Button
        label="Delete Item"
        click="deleteItem();"/>
    <mx:Button
        label="Add Item"
        click="addItem();"/>
</s:Application>
```

To use a MX data effect with the TileList class, apply the DefaultTileListEffect effect to the control. When an item in the TileList control is removed, this effect first fades out the item, and then moves the remaining items to their new position. When you add an item to the TileList control, this effect moves the existing items to their new position, and then fades in the new item.

You typically set the `offscreenExtraRowsOrColumns` property of the TileList control when you apply a data effect. This property specifies the number of extra rows or columns of offscreen item renderers used in the layout of the control. This property is useful because data effects work by first determining a *before* layout of the list-based control, then determining an *after* layout, and finally setting the properties of the effect to create an animation from the before layout to the after layout. Because many effects cause currently invisible items to become visible, or currently visible items to become invisible, this property configures the control to create the offscreen item renderers so that they already exist when the data effect plays.

You typically set the `offscreenExtraRowsOrColumns` property to a nonzero, even value, such as 2 or 4, for a TileList control, as the following example shows:

```
<?xml version="1.0"?>
<!-- dataEffects\TileListEffectCustomDefaultEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="500">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.effects.DefaultTileListEffect;
            import mx.effects.easing.Elastic;
            import mx.collections.ArrayCollection;
            import mx.effects.Move;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection(
                ["A","B",'C','D','E','F','G','H','I','J','K','L','M','N','O','P']);
            private function deleteItems():void {
                // As each item is removed, the index of the other items changes.
                // So first get the items to delete, and then determine their indices
                // as you remove them.
                var toRemove:Array = [];
                for (var i:int = 0; i < tlist0.selectedItems.length; i++)
                    toRemove.push(tlist0.selectedItems[i]);
                for (i = 0; i < toRemove.length; i++)
                    myDP.removeItemAt(myDP.getItemIndex(toRemove[i]));
            }
            private var zcount:int = 0;
            private function addItems():void {
                myDP.addItemAt("Z"+zcount++,Math.min(2,myDP.length));
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <!-- Define an instance of the DefaultTileListEffect effect,
             and set its moveDuration and color properties. -->
        <mx:DefaultTileListEffect id="myDTLE"
            moveDuration="100"/>
    </fx:Declarations>
    <mx:TileList id="tlist0"
        height="400" width="400"
        columnCount="4" rowCount="4"
        fontSize="30" fontWeight="bold"
        direction="horizontal"
        dataProvider="{myDP}"
        allowMultipleSelection="true"
        offscreenExtraRowsOrColumns="2"
        itemsChangeEffect="{myDTLE}" />
    <mx:Button
        label="Delete Selected Item(s)"
        click="deleteItems();"/>
    <mx:Button
        label="Add Item"
        click="addItems();"/>
</s:Application>
```

Improving performance when resizing Panel containers

When you apply a Resize effect to a MX Panel container, the measurement and layout algorithm for the effect executes repeatedly over the duration of the effect. When a Panel container has many children, the animation can be jerky because Flex cannot update the screen quickly enough.

To solve this problem, you can use the Resize effect's `hideChildrenTargets` property to hide the children of panel containers while the Resize effect is playing. The value of the `hideChildrenTargets` property is an Array of Panel containers that should include the Panel containers that resize during the animation. Before the Resize effect plays, Flex iterates through the Array and hides the children of each of the specified Panel containers.

In the following example, the children of the panelOne and panelTwo containers are hidden while the containers resize:

```
<?xml version="1.0"?>
<!-- behaviors\PanelResize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:Resize id="myResizeOne" heightTo="300"
            target="{panelOne}"
            hideChildrenTargets="{[panelOne]}"/>
        <mx:Resize id="myResizeTwo" heightTo="300"
            target="{panelTwo}"/>
    </fx:Declarations>
    <s:Button id="b1"
        label="Reset"
        click="panelOne.height=200;panelTwo.height=200;"/>
    <mx:Panel id="panelOne" title="Panel 1" height="200"
        mouseDown="myResizeOne.end();myResizeOne.play();">
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
    </mx:Panel>

    <mx:Panel id="panelTwo" title="Panel 2" height="200"
        mouseDown="myResizeTwo.end();myResizeTwo.play();">
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
        <s:Button label="Click Me"/>
    </mx:Panel>
</s:Application>
```

For the MX Panel container, you can control the effect used to hide the container children. For each panel container in the `hideChildrenTargets` Array, the following effect triggers execute:

- `resizeStartEffect` Delivered before the Resize effect begins playing.
- `resizeEndEffect` Delivered after the Resize effect finishes playing.

  If the `resizeStartEffect` trigger specifies an effect to play, the Resize effect is delayed until the effect finishes playing.

The default value for the MX Panel container's `resizeStartEffect` and `resizeEndEffect` triggers is `Dissolve`, which plays the Dissolve effect. To disable the Dissolve effect so that a Panel container's children are hidden immediately, you must set the value of the `resizeStartEffect` and `resizeEndEffect` triggers to `none`.

Using the MX Dissolve effect with the MX Panel, TitleWindow, and Accordion containers

The MX Dissolve effects is only applied to the content area of the Panel, TitleWindow, and Accordion containers. Therefore, the title bar of the Panel and TitleWindow containers, and the navigation buttons of the Accordion container are not modified by this effect.

To apply the Dissolve effect to the entire container, create a RoundedRectange instance that is the same size as the container, and then use the `targetArea` property of the effect to specify the area on which to apply the effect. In the following example, you apply a Dissolve effect to the first Panel container, and apply a Dissolve effect to a RoundedRectange instance overlaid on top of the second Panel container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- behaviors\PanelDissolve.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="init();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
           import mx.geom.*;
            // Define a bounding box for the target area of the effect.
           [Bindable]
           public var tArea:RoundedRectangle;
            // Size the bounding box to the size of Panel 2.
           private function init():void
           {
              tArea = new RoundedRectangle(0,0, panel2.width, panel2.height, 5);
           }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:Dissolve id="dissolveP1"
            duration="1000"
            target="{panel1}"
            alphaFrom="1.0" alphaTo="0.0"/>
        <!-- Apply the effect to the bounding box, not to Panel 2. -->
        <mx:Dissolve id="dissolveP2"
            duration="1000"
            target="{panel2}"
            alphaFrom="1.0" alphaTo="0.0"
            targetArea="{tArea}"/>
    </fx:Declarations>
    <mx:Panel id="panel1" title="Panel 1"
        width="100" height="140" >
       <mx:Button label="Orange" />
    </mx:Panel>
    <mx:Panel id="panel2" title="Panel 2"
        width="100" height="140" >
       <mx:Button label="Red" />
    </mx:Panel>
    <mx:Button label="Dissolve Panel 1"
        click="dissolveP1.end();dissolveP1.play();"/>
    <mx:Button label="Dissolve Panel 2"
        click="dissolveP2.end();dissolveP2.play();"/>
</s:Application>
```

Using MX easing functions

You can change the speed of an animation by defining an easing function for an effect. With easing, you can create a more realistic rate of acceleration and deceleration. You can also use an easing function to create a bounce effect or control other types of motion.

Flex supplies MX easing functions in the mx.effects.easing package. This package includes functions for the most common types of easing, including Bounce, Linear, and Sine easing. For more information on using these functions, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The following code shows the basic format of an easing function:

```
function myEasingFunction(t:Number, b:Number, c:Number, d:Number):Number {
    ...
}
```

You specify the following arguments of an easing function:

- `t` specifies time.
- `b` specifies the initial position of a component.
- `c` specifies the total change in position of the component.
- `d` specifies the duration of the effect, in milliseconds.

Configuring a MX easing function

You specify an easing function to a component by passing a reference to the function to a component property. You pass only the name of the easing function; Flex automatically sets the values for the arguments of the easing function.

All tween effects, meaning effect classes that are child classes of the TweenEffect class, support the `easingFunction` property, which lets you specify an easing function to the effect. Mask effects, those effect classes that are child classes of the MaskEffect class, also support easing functions. Other components support easing functions as well. For example, the Accordion and Tree components let you use the `openEasingFunction` style property to specify an easing function, and the ComboBox component supports a `closeEasingFunction` style property.

For example, you can specify the `mx.effects.easing.Bounce.easeOut()` method to the Accordion container using the `openEasingFunction` property, as the following code shows.

```
<?xml version="1.0"?>
<!-- behaviors\EasingFuncExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="550">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        import mx.effects.easing.*;
    </fx:Script>
    <mx:Accordion
        openEasingFunction="{Bounce.easeOut}"
        openDuration="2000">
        <mx:VBox label="Pane 1" width="400" height="400"/>
        <mx:VBox label="Pane 2" width="400" height="400"/>
    </mx:Accordion>
</s:Application>
```

Creating a custom easing function

In the following example, you create a custom easing function that creates a bounce motion when combined with the Flex Move effect:

```
<?xml version="1.0"?>
<!-- behaviors\Easing.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="650">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            private function myEasingFunction(t:Number, b:Number,
                    c:Number, d:Number):Number {
              if ((t /= d) < (1 / 2.75)) {
                  return c * (7.5625 * t * t) + b;
              }
              else if (t < (2 / 2.75)) {
                  return c * (7.5625 * (t-=(1.5/2.75)) * t + .75) + b;
              }
              else if (t < (2.5 / 2.75)) {
                  return c * (7.5625 * (t-=(2.25/2.75)) * t + .9375) + b;
              }
              else {
                  return c * (7.5625 * (t-=(2.625/2.75)) * t + .984375) + b;
              }
            };
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:Move id="moveLeftShow"
            xFrom="600" xTo="0" yTo="0"
            duration="3000"
            easingFunction="myEasingFunction"/>
        <mx:Move id="moveRightHide"
            xFrom="0" xTo="600"
            duration="3000"
            easingFunction="myEasingFunction"/>
    </fx:Declarations>

    <mx:LinkBar dataProvider="myVS"/>
    <mx:ViewStack id="myVS" borderStyle="solid">
        <mx:Canvas id="Canvas0" label="Canvas0"
            creationCompleteEffect="{moveLeftShow}"
            showEffect="{moveLeftShow}"
            hideEffect="{moveRightHide}" >
                <mx:Box height="300" width="600" backgroundColor="#00FF00">
                    <mx:Label text="Screen 0" color="#FFFFFF" fontSize="40"/>
                </mx:Box>
        </mx:Canvas>
        <mx:Canvas id="Canvas1" label="Canvas1"
            showEffect="{moveLeftShow}" hideEffect="{moveRightHide}" >
                <mx:Box height="300" width="600" backgroundColor="#0033CC">
                    <mx:Label text="Screen 1" color="#FFFFFF" fontSize="40"/>
                </mx:Box>
            </mx:Canvas>
    </mx:ViewStack>
</s:Application>
```

In this example, you use the custom effects in the `showEffect` and `hideEffect` properties of the children of a ViewStack container. When you click a label in the LinkBar control, the corresponding child of the ViewStack container slides in from the right, and bounces to a stop against the left margin of the ViewStack container, while the previously visible child of the ViewStack container slides off to the right.

The custom effect for the `showEffect` property is only triggered when the child's visibility changes from `false` to `true`. Therefore, the first child of the ViewStack container also includes a `creationCompleteEffect` property. This is necessary to trigger the effect when Flex first creates the component. If you omit the `creationCompleteEffect` property, you do not see the `moveLeftShow` effect when the application starts.

View states

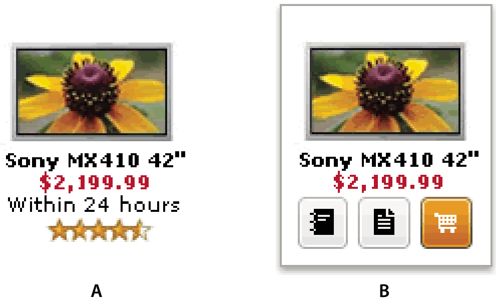
View states let you vary the content and appearance of a component or application, typically in response to a user action. When changing the view state, you can change the value of a property or style, change an event handler, or change the parent of a component.

Transitions define how a change of view state looks as it occurs on the screen. For information on transitions, see "Transitions" on page 1870.

About view states

In many rich Internet applications, the interface changes based on the task the user is performing. A simple example is an image that changes when the user rolls the mouse over it. More complex examples include user interfaces whose contents change depending on the user's progress through a task, such as changing from a browse view to a detail view. View states let you easily implement such applications.

At its simplest, a view state defines a particular view of a component. For example, a product thumbnail could have two view states; a default view state with minimal information, and a "rich" state with links for more information. The following figure shows two view states for a component:



A. *Default view state* ***B.*** *Rich view state*

To create a view state, you define a default view state, and then define a set of changes, or *overrides*, that modify the default view state to define the new view state. Each additional view state can modify the default view state by adding or removing child components, by setting style and property values, or by defining state-specific event handlers.

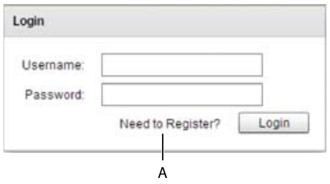
For example, the default view state of the application could be the home page and include a logo, a sidebar, and some welcome content. When the user clicks a button in the sidebar, the application dynamically changes its appearance, meaning its view state, by replacing the main content area with a purchase order form but leaving the logo and sidebar in place.

Two places in your application where you must use view states is when defining skins and item renderers for Spark components. For more information, see “Spark Skinning” on page 1602 and “Custom Spark item renderers” on page 470.

Adobe Flash Builder also has built-in support for view states. For more information on using Flash Builder, see Add View states and transitions.

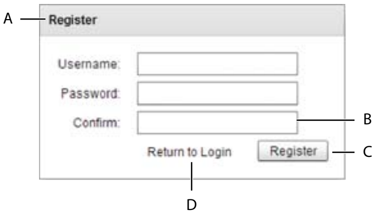
Defining a login interface by using view states

One use of view states is to implement a login and registration form. In this example, the default view state prompts the user to log in, and includes a LinkButton control that lets the user register, if necessary, as the following image shows:



LinkButton control (A)

If the user selects the Need to Register link, the form changes view state to display registration information, as the following image shows:



A. *Modified title of Panel container* ***B.*** *New form item* ***C.*** *Modified label of Button control* ***D.*** *New LinkButton control*

Notice the following changes to the default view state to create this view state:

- The title of the Panel container is set to Register
- The Form container has a new TextInput control for confirming the password
- The label of the Button control is set to Register
- The LinkButton control has been replaced with a new LinkButton control that lets the user change state back to the default view state

When the user clicks the Return to Login link, the view state changes back to the default view state to display the Login form. This change reverses all the changes made when changing to the register view state.

To see the code that creates this example, see “Example: Login form application” on page 1859.

Comparing view states to MX navigator containers

View states give you one way to change the appearance of an application or component in response to a user action. You can also use MX navigator containers, such as the Accordion, Tab Navigator, and ViewStack containers when you perform changes that affect several components.

Your choice of using navigator containers or states depends on your application requirements and user-interface design. For example, if you want to use a tabbed interface, use a TabNavigator container. You might decide to use the Accordion container to let the user navigate through a complex form, rather than using view states to perform this action.

When comparing view states to ViewStack containers, one thing to consider is that you cannot easily share components between the different views of a ViewStack container. That means you have to recreate a component each time you change views. For example, if you want to show a search component in all views of a View Stack container, you must define it in each view.

When using view states, you can easily share components across multiple view states by defining the component once, and then including it in each view state. For more information about sharing components among view states, see “Controlling when to create added children” on page 1856. For more information on navigator containers, see “MX navigator containers” on page 628.

Example: Creating a simple view state

The following example shows an application with a default view state, and one additional view state named "NewButton". In this example, changing to the NewButton view state adds button b2 to the Group container, and disables button b1. Switching back to the default view state removes b2 and enables b1:

```
<?xml version="1.0"?>
<!-- states\StatesSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:states>
        <!-- Define the new view states. -->
        <s:State name="default"/>
        <s:State name="NewButton"/>
    </s:states>
    <s:VGroup id="g1">
        <s:HGroup>
            <!-- Disable Button in the NewButton view state. -->
            <s:Button id="b1" label="Click Me"
                enabled.NewButton="false"/>
            <!-- Add a new child control to the VBox. -->
            <s:Button id="b2" label="New Button"
                includeIn="NewButton"/>
        </s:HGroup>

        <!-- Define Button control to switch to NewButton view state. -->
        <s:Button label="Change to NewButton state"
            click="currentState='NewButton';"/>
        <!-- Define Button control to switch to default view state. -->
        <s:Button label="Change to default view state"
            click="currentState='default';"/>
    </s:VGroup>
</s:Application>
```

Create and apply view states

The properties, styles, event handlers, and components that you define for an application or custom component specify its default view state. Any additional view state specify changes to the default view state.

For each additional view state, define overrides of the default view state. Overrides modify the properties, styles, and event handlers for a component, add a component to a view state, remove a component from a view state, or change the parent container of a component in a view state.

Creating view states

Consider the following when you define a view state.

- Define view states at the root of an application or at the root of a custom component; that is, as a property of the Application container of an application file, or of the root tag of an MXML component.
- Define states by using a component's `states` property, normally by using the `<s:states>` tag in MXML.
- Populate the `states` property with an Array of one or more State objects, where each State object corresponds to a view state.
- Use the `name` property of the State object to specify its identifier. To change to that view state, you set a component's `currentState` property to the value of the `name` property.

  ***Note:** View state definitions are processed at compile time. Therefore, you cannot use data binding to set the `State.name` property because data binding occurs at run time.*

The following MXML code shows this structure:

```
<s:Application>
    <!-- Define the view states.
        The <s:states> tag can also be a child tag of
        the root tag of a custom component.
    -->
    <s:states>
        <s:State name="State1"/>
        <s:State name="State2"/>
        <s:State name="State3"/>
        .
        .
    </s:states>
    <!-- Application definition. -->
    .
    .
</s:Application>
```

The default view state is defined as the first view state in the `<s:states>` Array. An application uses the default view state when it loads. The name of the default view state is not reserved, so it is not required to be "default".

Changing view state

The UIComponent class defines the `currentState` property that you use to set the current view state. When the application starts, the default value of the `currentState` property is the name of the first view state defined by the `<s:states>` tag.

In the next example, you use a Button control to set the `currentState` property of the Application object to "State1" or "State2", the names of a view state specified by the `<s:State>` tag:

```
<s:Button id="b1" label="Change to State 1" click="currentState='State2';"/>
<s:Button id="b2" label="Change to the default" click="currentState='State1';"/>
```

The second button in the previous example sets the current state to "State1" so that you can switch back to the default view state from "State2".

You can also set the `currentState` property to an empty String to set it to the default state, as the following example shows:

```
<s:Button id="b2" label="Change to the default" click="currentState='';"/>
```

You can change a component's view state by calling the `setCurrentState()` method of the UIComponent class. Use this method when you do *not* want to apply a transition that you have defined between two view states. For more information on transitions, see "Transitions" on page 1870.

Setting properties, styles, and events for a view state

Define state-specific property and style values by using the dot (.) operator with any writable MXML tag attribute. The dot notation has the following format:

`propertyOrStyleName.stateName`

For example, specify the value of the `label` property of a Button control for the default view state and for one additional view state, as the following example shows:

```
<s:Button label="Default State" label.State2="New State"/>
```

The value of the `label` property qualified by "State2" specifies the value for the property in that view state. The unqualified property definition, the one that omits the dot notation, defines the value of the property for all view states where you do not explicitly define an override.

Do not define an override for every property, style, and event for all view states. Only define the override for those view states where you want to change the value of the element.

You can also use child tags with the dot notation to define property overrides, as the following example shows for the b1 button control:

```
<?xml version="1.0"?>
<!-- states\StatesSimpleChildTags.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:states>
        <!-- Define the new view states. -->
        <s:State name="default"/>
        <s:State name="NewButton"/>
    </s:states>

    <s:VGroup >
        <s:HGroup>
            <!-- Disable Button in the NewButton view state. -->
            <s:Button id="b1">
                <s:label>Click Me</s:label>
                <s:label.NewButton>Disabled</s:label.NewButton>
                <s:enabled.NewButton>false</s:enabled.NewButton>
            </s:Button>

            <!-- Add a new child control to the VBox. -->
            <s:Button id="b2" label="New Button"
                includeIn="NewButton"/>
        </s:HGroup>
        <!-- Define Button control to switch to NewButton view state. -->
        <s:Button label="Change to NewButton state"
            click="currentState='NewButton';"/>
        <!-- Define Button control to switch to default view state. -->
        <s:Button label="Change to default view state"
            click="currentState='default';"/>
    </s:VGroup>
</s:Application>
```

To clear the value of a property in a view state, set the property to the value `@Clear()`, as the following example shows:

```
<s:Button color="0xFF0000" color.State1="@Clear()"/>
```

For a style property, setting the value to `@Clear()` corresponds to calling the `clearStyle()` method on the property.

Use the dot operator to change the event handler for a specific view state, as the following example shows. In this example, the handler for the `click` event for button b1 is set based on the view state:

```
<?xml version="1.0"?>
<!-- states\StatesEventHandlersSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:states>
        <s:State name="default"/>
        <s:State name="logout"/>
    </s:states>

    <s:VGroup >
        <s:Button id="b1" label="Click Me"
            click="ta1.text='hello';"
            click.logout="ta1.text='goodbye'"/>
        <s:TextArea id="ta1" height="100" width ="50%"/>

        <s:Button label="Default State"
            click="currentState='';"
            enabled="false"
            enabled.logout="true"/>
        <s:Button label="Logout State"
            click="currentState='logout';"
            enabled="true"
            enabled.logout="false"/>
    </s:VGroup>
</s:Application>
```

Adding or removing components for a view state

Use the `includeIn` and `excludeFrom` MXML attributes to specify the set of view states in which a component is included, where:

- The `includeIn` attribute specifies the list of view states where the component appears. This attribute takes a comma delimited list of view state names, all which must have been previously declared in the `<s:states>` Array.
- The `excludeFrom` attribute specifies the list of view states where the component is omitted. This attribute takes a comma delimited list of view state names.
- The `excludeFrom` and `includeIn` attributes are mutually exclusive; it is an error to define both on a single MXML tag.

The following example uses view states to add components to the application based on the current state:

```
<?xml version="1.0"?>
<!-- states\StatesSimpleIncExc.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:states>
        <s:State name="default"/>
        <s:State name="addCheckBox"/>
        <s:State name="addTextInput"/>
        <s:State name="addCheckBoxandButton"/>
    </s:states>

    <s:HGroup >
        <!-- Included in the addCheckBox and addCheckBoxandButton view states. -->
        <s:CheckBox id="myCB" label="Checkbox"
           includeIn="addCheckBox, addCheckBoxandButton"/>

        <!-- Included in the addTextInput view state. -->
        <s:TextInput id="myTI"
           includeIn="addTextInput"/>
        <!-- Included in the addCheckBoxandButton view state. -->
        <s:Button id="myB"
           includeIn="addCheckBoxandButton"/>

        <!-- Exclude from addTextInput view state. -->
        <s:TextArea text="Exclude from addTextInput"
            excludeFrom="addTextInput"/>
    </s:HGroup>
    <s:HGroup>
        <s:Button label="Add CheckBox"
            click="currentState='addCheckBox'"/>
        <s:Button label="Show Textinput Only"
            click="currentState='addTextInput'"/>
        <s:Button label="Add CheckBox and Button"
            click="currentState='addCheckBoxandButton'"/>
        <s:Button label="Default"
            click="currentState='default'"/>
    </s:HGroup>
</s:Application>
```

You can specify the `includeIn` and `excludeFrom` attributes on any MXML object within an MXML document, with the exception of the following tags:

- The root tag of an MXML document, such as the Application tag or the root tag in an MXML component.
- Tags that represent properties of their parent tag. For example, the `<s:label>` tag of the `<s:Button>` tag.
- Descendants of the `<fx:XML>`, `<fx:XMLList>`, or `<fx:Model>` tags.
- Any language tags, such as the `<fx:Binding>`, `<fx:Declarations>`, `<fx:Metadata>`, `<fx:Script>`, and `<fx:Style>` tags.

Restriction on modifying container children when using view states

When you use view states to control the children of a container, do not add or remove container children, or change the order of children in the container, at runtime. For example, the following container defines a Button control that appear only in State2:

```
<s:Group id="myGroup">
    <s:Button/>
    <s:Button includeIn="State2"/>
    <s:Button/>
</s:Group>
```

The view states infrastructure relies on the structure of the container as defined by the MXML file. Changing the child order of the container at runtime can cause your application to fail. Therefore, do not call the `addElement()`, `removeElement()`, `setElementIndex()` method on the Group container at runtime, or any other method that changes the child order of the container.

Changing the parent of a component

A view state can change the parent container of a component. Changing the parent container of a component is called *reparenting* the component.

Use the `<fx:Reparent>` tag to change the parent container of a component. The `<fx:Reparent>` tag has the following syntax:

```
<fx:Reparent target="targetComp" includeIn="stateName">
```

The `target` property specifies the target component, and the `includeIn` property specifies a view state. When the current view state is set to *stateName*, the target component becomes a child of the parent component of the `<fx:Reparent>` tag. You can think of the `<fx:Reparent>` tag as a placeholder for the component for a specific view state.

You can use the `<fx:Reparent>` tag in any component that can hold a child component, and the child can use the `includeIn` or `excludeFrom` keywords.

The following example uses the `<fx:Reparent>` tag to switch a Button control between two Panel containers:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- states\NewStatesReparent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:states>
        <s:State name="Parent1"/>
        <s:State name="Parent2"/>
    </s:states>

    <s:HGroup>
        <s:Panel id="Panel1"
            height="100" width="100"
            title="Panel 1">
            <s:Button id="setCB" includeIn="Parent1"/>
        </s:Panel>
        <s:Panel id="Panel2"
            height="100" width="100"
            title="Panel 2">
            <fx:Reparent target="setCB" includeIn="Parent2"/>
        </s:Panel>
    </s:HGroup>

    <s:HGroup>
        <s:Button label="Parent 1"
            click="currentState='Parent1'"
            enabled.Parent1="false"/>
        <s:Button label="Parent 2"
            click="currentState='Parent2'"
            enabled.Parent2="false"/>
    </s:HGroup>
</s:Application>
```

Two `<fx:Reparent>` tags cannot target the same component for the same view state. That means a component can only have a single parent in each view state.

Controlling when to create added children

When you remove a component as part of a change of view state, you remove the component from the application's display list, which means that it no longer appears on the screen. Even though the component is no longer visible, the component still exists and you can access it from within your application.

When you add a component as part of a change of view state, you can either create the component before the first change to the view state, or create it at the time of the first change. If you create the component before the first change, you can access the component from within your application even though you have not yet changed view states. If you create the component when you perform the first change to the view state, you cannot access it until you perform that first change.

By default Flex creates container children when they are first required as part of a change of view state. However, if a child requires a long time to create, users might see a delay when the view state changes. Therefore, you can choose to create the child before the state changes to improve your application's apparent speed and responsiveness.

Regardless of when you create a component, the component remains in memory after you change out of the view state that creates it. Therefore, after the first change to a view state, you can always access the component even if that view state is no longer the current view state.

The specification of when the child is created is called its *creation policy*. For more general information on creation policies and controlling how children are created, see "Improving startup performance" on page 2333.

Use the `itemCreationPolicy` property to specify the creation policy. The `itemCreationPolicy` property supports the following values:

`deferred` Creates the component instance when it is first added by a change to the view state. This is the default value.

`immediate` Creates the component instance at application startup.

The following example uses the Immediate view state to add a Button control named newButtonImmediate. You set the `itemCreationPolicy` property for this button to `immediate` to create the button at application startup. Therefore, the application can access it to set its `label` property before it switches to the Immediate view state.

The application also uses the Deferred view state to create the button named newButtonDeferred with the `itemCreationPolicy` property set to `deferred`. Therefore, this button is created when you first change to the Deferred view state, and you cannot access it until after the first switch to the Deferred view state:

```
<?xml version="1.0"?>
<!-- states\StatesCreationPolicy.mxml  -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initButton();">

    <fx:Script>
        <![CDATA[
            // Because the Immediate view state creates the Button control
            // at application startup, you can access the control to
            // set the label before the first switch
            // to the Immediate view state.
            public function initButton():void {
                newButtonImmediate.label="Immediate Button";
                // Uncommenting this line to access the label causes a
                // Run Time Exception because newButtonDeferred does not exist yet.
                // newButtonDeferred.label="Deferred Button";
            }
        ]]>
    </fx:Script>
    <s:states>
        <s:State name="default"/>
        <s:State name="Immediate"/>
        <s:State name="Deferred"/>
    </s:states>

    <s:Panel id="myPanel"
        title="Static and dynamic states"
        width="300" height="150">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <!-- Create the Button control at application startup. -->
        <s:Button id="newButtonImmediate"
```

```
            includeIn="Immediate"
            itemCreationPolicy="immediate"/>
        <!-- Create the Button control when you switch to this view state. -->
        <s:Button id="newButtonDeferred"
            label="Deferred button"
            includeIn="Deferred"
            itemCreationPolicy="deferred"/>

        <!-- Change to the Immediate view state. -->
        <s:Button label="Change to Immediate state"
            click="currentState='Immediate';"/>

        <!-- Change to the Deferred view state. -->
        <s:Button label="Change to Deferred state"
          click="currentState='Deferred';"/>
        <!-- Change to the default view state. -->
        <s:Button label="Change to default state"
          click="currentState='default';"/>
    </s:Panel>
</s:Application>
```

Controlling caching of objects created in a view state

A change to a view state can cause Flex to create an object. By default, after Flex creates the object, the object is cached indefinitely, even after you switch to another view state. The item is cached even if the destination view state excludes the object.

Use the `itemDestructionPolicy` attribute with the `includeIn` and `excludeFrom` attributes configure Flex to completely destroy the object when leaving a view state, including deleting it from the cache. Typically it is more efficient to allow items to be cached because it improves performance when switching back to the view state that created the object, or to a view state that uses the object. However your application might define a view state that is rarely used, and you do not want to allocate memory for caching the object.

You can use the `itemDestructionPolicy` attribute on any object in MXML that supports the `includeIn` and `excludeFrom` attributes. The possible values for `itemDestructionPolicy` are `never` (default) and `auto`.

The value of `never` specifies that the object is cached indefinitely. A value of `auto` means that the object is destroyed when leaving a view state where the object exists. The following example sets the attribute to `auto`:

```
<s:TextInput includeIn="newTextInput" itemDestructionPolicy="auto"/>
```

Defining view state groups

The `stateGroups` attribute of the `<s:States>` tag lets you group one or more states together. For example, if multiple components appear in the same set of view states, create a view state group that contains all these view states. Then, when you set the `currentState` property to any view state in the group, the components appears.

In the following example, the CheckBox control named myCB appears when you change to any view state in Group1:

```
<?xml version="1.0"?>
<!-- states\StatesSimpleGroups.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:states>
        <s:State name="default"/>
        <s:State name="addCheckBox" stateGroups="Group1"/>
        <s:State name="addTextInput"/>
        <s:State name="addCheckBoxandButton" stateGroups="Group1"/>
    </s:states>

    <s:HGroup>
        <!-- Included in the addCheckBox and addCheckBoxandButton view states. -->
        <s:CheckBox id="myCB" label="Checkbox"
            includeIn="Group1"/>

       <!-- Included in the addTextInput view state. -->
       <s:TextInput id="myTI"
            includeIn="addTextInput"/>
        <!-- Included in the addCheckBoxandButton view state. -->
        <s:Button id="myB"
           includeIn="addCheckBoxandButton"/>

        <!-- Exclude from addTextInput view state. -->
        <s:TextArea text="Exclude from addTextInput"
           excludeFrom="addTextInput"/>
    </s:HGroup>
    <s:HGroup>
        <s:Button label="Add CheckBox"
            click="currentState='addCheckBox'"/>
        <s:Button label="Add Textinput"
            click="currentState='addTextInput'"/>
        <s:Button label="Add Group 1"
            click="currentState='addCheckBoxandButton'"/>
        <s:Button label="Default"
            click="currentState='default'"/>
    </s:HGroup>
</s:Application>
```

Example: Login form application

The following example creates the Login and Register forms shown in “About view states” on page 1847. This application has the following features:

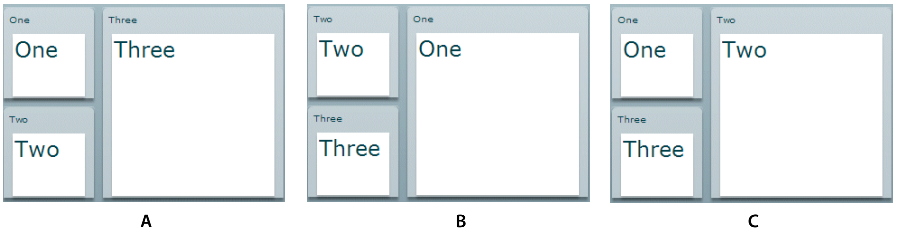
- When the user clicks the Need to Register LinkButton control, the event handler for the `click` event sets the view state to Register.
- The Register state code adds a TextInput control, changes properties of the Panel container and Button control, removes the existing LinkButton controls, and adds a new LinkButton control.
- When the user clicks the Return to Login LinkButton control, the event handler for the `click` event resets the view state to the default view state.

***Note:** For an example that adds a transition to animate the change between view states, see "Example: Using transitions with a login form" on page 1871.*

```
<?xml version="1.0"?>
<!-- states\LoginExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!-- The Application class states property defines the view states.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="Register"/>
    </s:states>
    <!-- Set title of the Panel container based on the view state.-->
    <s:Panel id="loginPanel"
        title="Login" title.Register="Register">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Form id="loginForm">
            <s:FormItem label="Username:">
                <s:TextInput/>
            </s:FormItem>

            <s:FormItem label="Password:">
                <s:TextInput/>
            </s:FormItem>

            <s:FormItem id="confirm" label="Confirm:" includeIn="Register">
             <!-- Add a TextInput control to the form for the Register view state. -->
                <s:TextInput/>
            </s:FormItem>

            <s:FormItem>
                <!-- Use the LinkButton to change view state.-->
                <s:HGroup>
                    <!-- Set label of the control based on the view state.-->
                    <mx:LinkButton id="registerLink"
                        label="Need to Register?"
                        label.Register="Return to Login"
                        click="currentState='Register'"
                        click.Register="currentState=''"/>
                    <s:Button id="loginButton"
                        label="Login" label.Register="Register"/>
                </s:HGroup>
            </s:FormItem>
        </s:Form>
    </s:Panel>
</s:Application>
```

Example: Controlling layout using view states groups

In this example, you define an application with three Panel containers and three view states, as the following example shows:



***A.** Default view state* ***B.** One view state* ***C.** Two view state*

To change view state, click the Panel container that you want to display in the expanded size. For a version of this example that adds a transition to animate the view state change, see “Defining transitions” on page 1872.

```
<?xml version="1.0"?>
<!-- states/ThreePanel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" width="400">
    <!-- Define the two view states, in addition to the default view state.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="One"/>
        <s:State name="Two"/>
    </s:states>
    <!-- Define the Group container holding the three Panel containers.-->
    <s:Group width="100%" height="100%">
        <s:Panel id="p1" title="One"
                x="0" y="0"
                x.One="110" y.One="0"
                x.Two="0" y.Two="0"
                width="100" height="100"
                width.One="200" height.One="210"
                width.Two="100" height.Two="100"
                click="currentState='One'">
            <s:Label fontSize="24" text="One"/>
        </s:Panel>

        <s:Panel id="p2" title="Two"
                x="0" y="110"
                x.One="0" y.One="0"
                x.Two="110" y.Two="0"
                width="100" height="100"
                width.One="100" height.One="100"
                width.Two="200" height.Two="210"
                click="currentState='Two'">
            <s:Label fontSize="24" text="Two"/>
        </s:Panel>

        <s:Panel id="p3" title="Three"
                x="110" y="0"
                x.One="0" y.One="110"
                x.Two="0" y.Two="110"
                width="200" height="210"
                width.One="100" height.One="100"
                width.Two="100" height.Two="100"
                click="currentState='default'">
            <s:Label fontSize="24" text="Three"/>
        </s:Panel>
    </s:Group>
</s:Application>
```

You can optimize this application by using view state groups, as the following example shows:

```
<?xml version="1.0"?>
<!-- states/ThreePanelGroups.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" width="400">
    <!-- Define the two view states, in addition to the default state.-->
    <s:states>
        <s:State name="default" stateGroups="grpDefaultOne, grpDefaultTwo"/>
        <s:State name="One" stateGroups="grpDefaultOne, grpOneTwo "/>
        <s:State name="Two" stateGroups="grpDefaultTwo, grpOneTwo"/>
    </s:states>
    <!-- Define the Group container holding the three Panel containers.-->
    <s:Group width="100%" height="100%">
        <s:Panel id="p1" title="One"
                x.grpDefaultTwo="0" y.grpDefaultTwo="0"
                x.One="110" y.One="0"
                width.grpDefaultTwo="100" height.grpDefaultTwo="100"
                width.One="200" height.One="210"
                click="currentState='One'">
            <s:Label fontSize="24" text="One"/>
        </s:Panel>
        <s:Panel id="p2" title="Two"
                x="0" y="110"
                x.One="0" y.One="0"
                x.Two="110" y.Two="0"
                width.grpDefaultOne="100" height.grpDefaultOne="100"
                width.Two="200" height.Two="210"
                click="currentState='Two'">
            <s:Label fontSize="24" text="Two"/>
        </s:Panel>

        <s:Panel id="p3" title="Three"
                x="110" y="0"
                x.grpOneTwo="0" y.grpOneTwo="110"
                width="200" height="210"
                width.grpOneTwo="100" height.grpOneTwo="100"
                click="currentState='default'">
            <s:Label fontSize="24" text="Three"/>
        </s:Panel>
    </s:Group>
</s:Application>
```

Using view state events

When a component's `currentState` property changes, the State object for the states being exited and entered dispatch the following events:

enterState Dispatched after a view state is entered. Dispatched by a State object and by a component.

exitState Dispatched when a view state is about to be exited. Dispatched by a State object before it is exited, and by a component before it exits the current view state.

The component on which you modify the `currentState` property to cause the state change dispatches the following events:

currentStateChanging Dispatched when the view state is about to change. It is dispatched by a component after its `currentState` property changes, but before the view state changes. You can use this event to request any data from the server required by the new view state.

currentStateChange Dispatched after the view state has completed changing. It is dispatched by a component after its `currentState` property changes. You can use this event to send data back to a server indicating the user's current view state.

The following example uses the `enterState` and `exitState` events to update two TextArea controls with the name of the new state and of the old state:

```
<?xml version="1.0"?>
<!-- states\StatesSimpleEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="450">
    <s:states>
        <!-- Define the new view states. -->
        <s:State name="default"
            enterState="MyEnterTA.text = 'Enter state: default';"
            exitState="MyExitTA.text = 'Exit state: default';"/>
        <s:State name="NewButton"
            enterState="MyEnterTA.text = 'Enter state: NewButton';"
            exitState="MyExitTA.text = 'Exit state: NewButton';"/>
    </s:states>
    <s:VGroup id="g1">
        <s:HGroup>
            <s:Button id="b1" label="Click Me"
                enabled.NewButton="false"/>
            <s:Button id="b2" label="New Button"
                includeIn="NewButton"/>
        </s:HGroup>
        <s:Button label="Change to NewButton state"
            click="currentState='NewButton';"/>
        <s:Button label="Change to default view state"
            click="currentState='default';"/>
        <s:TextArea id="MyEnterTA"/>
        <s:TextArea id="MyExitTA"/>

    </s:VGroup>
</s:Application>
```

Defining view states in custom components

If a custom component has multiple view states, define the view states in the component code, not in the main application. You can then set the view state of the component independently of the view state of the enclosing application.

The following example shows a custom component that implements a login dialog box with two view states: Login and Register. The main application sets the `currentState` property of the component to Login. However, you can set it to Register if you want that to be the default view state, or use data binding to set it at run time:

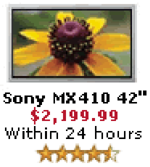
```
<?xml version="1.0"?>
<!-- states\myComponents\LoginComponent.mxml -->
<!-- Set title of the Panel container based on the view state.-->
<s:Panel xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    title="Login" title.Register="Register">

    <!-- The states property defines the view states.-->
    <s:states>
        <s:State name="Login"/>
        <s:State name="Register"/>
    </s:states>
    <s:Form id="loginForm">
        <s:FormItem label="Username:">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="Password:">
            <s:TextInput/>
        </s:FormItem>
        <!-- Add a TextInput control to the form for the Register view state. -->
        <s:FormItem id="confirm" label="Confirm:" includeIn="Register">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem>
            <s:HGroup>
              <!-- Use the LinkButton to change to the Register view state.-->
               <!-- Exclude the LinkButton from the Register view state.-->
                <mx:LinkButton id="registerLink"
                    includeIn="Login"
                    label="Need to Register?"
                    click="currentState='Register'"/>
                <!-- Add a LinkButton to the form for the Register view state. -->
                <mx:LinkButton label="Return to Login"
                    includeIn="Register"
                    click="currentState=''"/>
                <mx:Spacer width="100%" id="spacer1"/>
                <!-- Set label of the control based on the view state.-->
                <s:Button id="loginButton"
                    label="Login" label.Register="Register" />
            </s:HGroup>
        </s:FormItem>
    </s:Form>
</s:Panel>
```

You can then use this component in an application, as the following example shows:

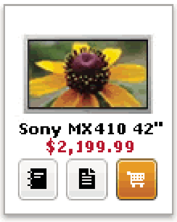
```
<?xml version="1.0" encoding="utf-8"?>
<!-- states\LoginMain.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Label text="Login or Register"
        fontSize="14" fontWeight="bold"/>

    <MyComp:LoginComponent currentState="Login"/>
</s:Application>
```

Using view states with a custom item renderer

A shopping application that displays multiple items on a page might have a custom thumbnail item renderer with two view states. In one view state, the item cell might look the following image:



When the user rolls the mouse over the item, the view state changes: the thumbnail no longer has the availability and rating information, but now has buttons that let the user get more information or add the item to the wish list or cart. In the new state, the cell also has a border and a drop shadow, as the following image shows:



In this example, the application item renderer's two view states have different child components and have different component styles. The summary state, for example, includes an availability label and a star rating image, and has no border. The rolled-over state replaces the label and rating components with three buttons, and has an outset border.

For information on item renderers, see "MX item renderers and item editors" on page 1006.

Example: Using view states with a custom item renderer

The following code shows an application that uses a custom item renderer to display catalog items. When the user moves the mouse over an item, the item renderer changes to a state where the picture is slightly enlarged, the price appears in the cell, and a text box shows a message about the item. All changes are made by the item renderer's state, including the change in the parent application.

```
<?xml version="1.0"?>
<!-- states\StatesRendererMain.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout horizontalAlign="center"/>
    </s:layout>

    <s:DataGroup itemRenderer="myComponents.ImageComp"
        width="300" height="400">
        <s:layout>
            <s:TileLayout columnWidth="150" rowHeight="175"
                requestedColumnCount="2" requestedRowCount="2" />
        </s:layout>
        <mx:ArrayCollection>
            <fx:Object
                name="Nokia 3595"
                data="1"
                price="129.99"
                image="assets/Nokia_3595.gif"
                description="Kids love it."/>
            <fx:Object
                name= "Nokia 3650"
                data="1"
                price="99.99"
                image="assets/Nokia_3650.gif"
                description="Impress your friends."/>
            <fx:Object
                name="Nokia 6010"
                data="1"
                price="49.99"
                image="assets/Nokia_6010.gif"
                description="Good for everyone."/>
            <fx:Object
                name="Nokia 6360"
                data="1"
                price="19.99"
                image="assets/Nokia_6360.gif"
                description="Great deal!"/>
        </mx:ArrayCollection>
    </s:DataGroup>
</s:Application>
```

The following code defines the item renderer, in the file ImageComp.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:states>
        <s:State name="normal"/>
        <s:State name="hovered"/>
        <s:State name="selected"/>
    </s:states>
    <s:Image id="img1"
        source="{data.image}"
        width="75" width.hovered="85"
        height="75" height.hovered="85"/>
    <s:Label text="{data.name}"
        color="blue"
        fontSize.hovered="16"/>
    <s:Label text.hovered="{data.price}"/>
    <s:TextArea id="t1"
        visible="false" visible.hovered="true"
        height="30" width="125"
        text.hovered="{data.description}"/>
</s:ItemRenderer>
```

Using view states with the browser manager

The Flex browser manager lets users navigate through an application by using the web browser's back and forward navigation commands. The browser manager can track when the application enters a state so that users can use the browser to navigate between states, such as states that correspond to different stages in an application process.

To use the browser manager, you first obtain a reference to the BrowserManager object by calling the `BrowserManager.getInstance()` method. This method returns the current instance of the manager, which implements IBrowserManager interface. You can then call methods on the browser manager such as `setTitle()` and `setFragment()`.

To save the current state of an application, you write the state name as a name-value pair to the application URL in the browser's address bar. When the user clicks the browser's Forward or Back button, extract the state from the browser's URL and set the `currentState` value appropriately. For more information the browser manager, see "Using the BrowserManager" on page 2024.

The following code shows how you can code a search interface so that the browser manager tracks the state of the application:

```
<?xml version="1.0"?>
<!-- states\StatesBrowserManager.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">

    <fx:Script>
        <![CDATA[
            import mx.managers.BrowserManager;
            import mx.managers.IBrowserManager;
            import mx.events.BrowserChangeEvent;
            import mx.utils.URLUtil;

            // The search string value.
            [Bindable]
            public var searchString:String;
            // The BrowserManager instance.
            private var browserManager:IBrowserManager;

            // Initialize the BrowserManager when the application is created.
            public function initApp():void {
                browserManager = BrowserManager.getInstance();
                browserManager.addEventListener(BrowserChangeEvent.BROWSER_URL_CHANGE, 
parseURL);
                browserManager.init("", "Browser Manager for View States");
                // Set the default state as a name/value pair in the URL.
                updateURL('default');
            }

            // Handle the event when the user clicks the Forward or Back button
            // in the browser. This event handler retrieves the value of the
            // state property from the new URL, and uses it to set currentState property.
            private var stateFromURL:String;
            private function parseURL(e:Event):void {
                var o:Object = URLUtil.stringToObject(browserManager.fragment);
                stateFromURL = o.state;
                currentState=stateFromURL;
            }
            // On a state change, save the value of the new
            // currentState property as name/value pair of the URL.
            private function updateURL(myCurrentState:String):void {
                var s:String = "state=" + myCurrentState;
                browserManager.setFragment(s);
            }

            // The method for doing the search.
            // For the sake of simplicity it doesn't do any searching.
            // It does change the state to display the results box,
            // and save the new state in the URL.
            public function doSearch():void {
                currentState = "results";
                updateURL('results');
                searchString = searchInput.text;
            }
            // Method to revert the state to the default state.
            // Saves the new state in in the URL.
            public function reset():void {
                currentState = '';
                searchInput.text = "";
                searchString = "";
                updateURL('default');
            }
        ]]>
    </fx:Script>

    <s:states>
        <!-- The state for displaying the search results -->
        <s:State name="default"/>
        <s:State name="results"/>
    </s:states>
    <!-- In the default state, just show a panel
        with a search text input and button. -->
    <s:Panel id="panel1"
        title="Search" title.results="Results"
        resizeEffect="Resize"
        width="10%" height="10%"
        width.results="100%" height.results="100%">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:SkinnableContainer id="searchFields" defaultButton="{b}">
            <s:layout>
                <s:HorizontalLayout/>
            </s:layout>
            <s:TextInput id="searchInput"/>
            <s:Button id="b"
                label="Go"
                click="doSearch();"/>
            <s:Button includeIn="results"
                label="Reset"
                click="reset();"/>
        </s:SkinnableContainer>
        <s:Label includeIn="results"
            text="Search results for {searchString}"/>
    </s:Panel>
</s:Application>
```

In this example, when the user changes state, the `updateURL()` method writes the current state to the browser's URL as a name/value pair. When the user clicks the browser's Forward or Back button, the `parseURL()` method extracts the state from the browser's URL and set the `currentState` value.

You can modify the `updateURL()` method to write more than just the current state to the URL. For example, you can write the search string to the URL. You then use the `parseURL()` method to extract it and update the TextInput control named searchInput.

Transitions

View states let you change the appearance of an application, typically in response to a user action. Transitions define how a change of view state looks as it occurs on the screen. You define a transition by using the effect classes, in combination with several effects designed explicitly for handling transitions.

To use transitions, you should be familiar with how effects and view states work. For more information on effects, see "Introduction to effects" on page 1784. For more information on view states, see "View states" on page 1847.

About transitions

View states let you vary the content and appearance of an application, typically in response to a user action. To use view states, you define the default view state of an application, and one or more additional view states.

When you change view states, Adobe® Flex® performs all the visual changes to the application at the same time. That means when you resize, move, or in some other way alter the appearance of the application, the application appears to jump from one view state to the next.

Instead, you might want to define a smooth visual change from one view state to the next, in which the change occurs over a period of time. A *transition* is one or more effects grouped together to play when a view state change occurs.

For example, your application defines a form that in its default view state shows only a few fields, but in an expanded view state shows additional fields. Rather than jumping from the base version of the form to the expanded version, you define a transition that uses the Resize effect to expand the form, and then uses the Fade effect to slowly make the new form elements appear on the screen.

When a user changes back to the base version of the form, your transition uses a Fade effect to make the fields of the expanded form disappear, and then uses the Resize effect to slowly shrink the form back to its original size.

By using a transition, you can apply one or more effects to the form, to the fields added to the form, and to fields removed from the form. You can apply the same effects to each form field, or apply different effects. You can also define one set of effects to play when you change state to the expanded form, and a different set of effects to play when changing from the expanded state back to the base state.

Example: Using transitions with a login form

The topic "View states" on page 1847 contains an example of a login form with two states: a base state and a register state. The following example adds a transition to this form to use a Wipe effect and a Resize effect with an easing function to animate the transition from view state to view state:

```
<?xml version="1.0" ?>
<!-- transitions\LoginFormTransition.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <!-- The Application class states property defines the view states.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="Register"/>
    </s:states>

    <!-- Define the transition to animate the change of view state. -->
    <s:transitions>
       <s:Transition fromState="default">
            <s:Parallel>
                <s:Wipe direction="right" target="{confirm}"/>
                <s:Resize target="{loginPanel}" duration="100"/>
            </s:Parallel>
        </s:Transition>
       <s:Transition fromState="Register">
            <s:Sequence>
```

```
                <s:Resize target="{loginPanel}" duration="100"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>

    <!-- Set title of the Panel container based on the view state.-->
    <s:Panel id="loginPanel"
        title="Login" title.Register="Register">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Form id="loginForm">
            <s:FormItem label="Username:">
                <s:TextInput/>
            </s:FormItem>
            <s:FormItem label="Password:">
                <s:TextInput/>
            </s:FormItem>
            <!-- Add a TextInput control to the form for the Register view state. -->
            <s:FormItem id="confirm" label="Confirm:" includeIn="Register">
                <s:TextInput id="myTI"/>
            </s:FormItem>
            <s:FormItem>
                <!-- Use the LinkButton to change to the Register view state.-->
                <!-- Exclude the LinkButton from the Register view state.-->
                <!-- Add a LinkButton to the form for the Register view state. -->
                <mx:LinkButton label="Return to Login"
                    includeIn="Register"
                    click="currentState=''"/>
                <mx:LinkButton id="registerLink"
                    includeIn="default"
                    label="Need to Register?"
                    click="currentState='Register'"/>
                <s:Button id="loginButton"
                    label="Login" label.Register="Register"/>
            </s:FormItem>
        </s:Form>
    </s:Panel>
</s:Application>
```

Comparing transitions to effects

Transitions do not replace effects; that is, you can still apply a single effect to a component. Transitions give you the ability to group multiple effects so that the effects are triggered together as part of a change of view states. Transitions are designed to work specifically with a change between view states, because a view state change typically means multiple components are modified at the same time.

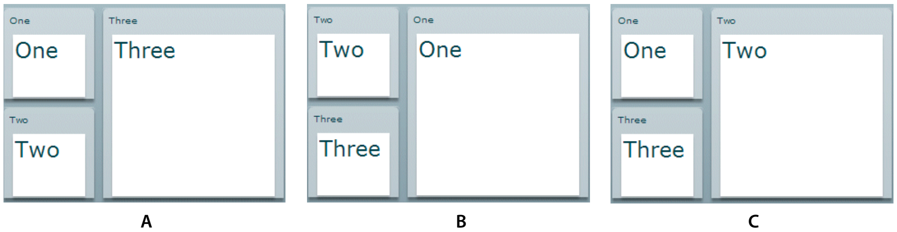
Transitions also support effect filters. A *filter* lets you conditionalize an effect so that it plays an effect only when the effect target changes in a certain way. For example, as part of the change to the view state, one or more components may change size. You can use a filter with an effect so that you apply the effect only to the components being resized. For an example using a filter, see “Filtering effects” on page 1882.

Defining transitions

You use the Transition class to create a transition. The following table defines the properties of the Transition class:

Property	Definition
`fromState`	A String that specifies the view state that you are changing from when you apply the transition. The default value is an asterisk, `"*"`, which means any view state.
`toState`	A String that specifies the view state that you are changing to when you apply the transition. The default value is an asterisk, `"*"`, which means any view state.
`effect`	The Effect object to play when you apply the transition. Typically, this is a composite effect, such as the Parallel or Sequence effect, that contains multiple effects.

You can define multiple Transition objects in an application. The UIComponent class includes a `transitions` property that you set to an Array of Transition objects. For example, you define an application with three Panel containers and three view states, as the following example shows:



A. Default view state B. One view state C. Two view state

You define the transitions for this example in MXML using the `<s:transitions>` tag, as the following example shows:

```
<?xml version="1.0" ?>
<!-- transitions/DefiningTrans.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" width="400" >
    <!-- Define the two view states, in addition to the base state.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="One"/>
        <s:State name="Two"/>
    </s:states>
    <!-- Define Transition array with one Transition object.-->
    <s:transitions>
        <!-- A transition for changing from any state to any state. -->
        <s:Transition id="myTransition" fromState="*" toState="*">
            <!-- Define a Parallel effect as the top-level effect.-->
            <s:Parallel id="t1" targets="{[p1,p2,p3]}">
                <!-- Define a Move and Resize effect.-->
                <s:Move  duration="400"/>
                <s:Resize duration="400"/>
            </s:Parallel>
        </s:Transition>
    </s:transitions>
    <!-- Define the container holding the three Panel containers.-->
    <s:Group id="pm" width="100%" height="100%">
        <s:Panel id="p1" title="One"
                x="0" y="0"
                x.One="110" y.One="0"
                x.Two="0" y.Two="0"
                width="100" height="100"
                width.One="200" height.One="210"
                width.Two="100" height.Two="100"
                click="currentState='One'">
            <s:Label fontSize="24" text="One"/>
        </s:Panel>

        <s:Panel id="p2" title="Two"
                x="0" y="110"
                x.One="0" y.One="0"
                x.Two="110" y.Two="0"
                width="100" height="100"
                width.One="100" height.One="100"
                width.Two="200" height.Two="210"
                click="currentState='Two'">
            <s:Label fontSize="24" text="Two"/>
        </s:Panel>

        <s:Panel id="p3" title="Three"
                x="110" y="0"
                x.One="0" y.One="110"
                x.Two="0" y.Two="110"
                width="200" height="210"
                width.One="100" height.One="100"
                width.Two="100" height.Two="100"
                click="currentState='default'">
            <s:Label fontSize="24" text="Three"/>
        </s:Panel>
    </s:Group>
</s:Application>
```

In this example:

1 You use the `click` event of each Panel container to change the view state.

2 When the application changes view state, Flex searches for a Transition object that matches the current and destination view state. In this example, you set the `fromState` and `toState` properties to `"*"`. Therefore, Flex applies myTransition to all state changes. For more information on setting the `fromState` and `toState` properties, see “Defining multiple transitions” on page 1875.

3 After Flex determines the transition that matches the change of view state, Flex applies the effects defined by the transition to the effect targets. In this example, you use the specification of the top-level Parallel effect in the transition to specify the targets as the three Panel containers. For more information on setting the effect targets, see “Defining effect targets” on page 1875.

The effects play in parallel on all effect targets, so the three Panel containers move to a new position and change size simultaneously. You can also define the top-level effect as a Sequence effect to make the effects occur sequentially, rather than in parallel.

Flex determines the start and end property values of the Move and Resize effect by using information from any properties that you specified to the effect, the current view state, and the destination view state. In this example, you omit any property specifications in the effect definitions, so Flex uses the current size and position of each Panel container to determine the values of the `Move.xFrom`, `Move.yFrom`, `Resize.widthFrom`, and `Resize.heightFrom` properties. Flex uses the destination view state to determine the values of the `Move.xTo`, `Move.yTo`, `Resize.widthTo`, and `Resize.heightTo` properties. For more information, see “Defining the effect start and end values” on page 1876.

Defining multiple transitions

You can define multiple transitions in your application so that you can associate a specific transition with a specific change to the view state. To specify the transition associated with a change to the view states, you use the fromState and toState properties.

By default, both the `fromState` and `toState` properties are set to `"*"`, which indicates that the transition should be applied to any changes in the view state. You can set either property to an empty string, `""`, which corresponds to the default view state.

You use the `fromState` property to explicitly specify the view state that your are changing from, and the `toState` property to explicitly specify the view state that you are changing to, as the following example shows:

```
<s:transitions>
    <!-- Play for a change to the login view state from any view state. -->
    <s:Transition id="toLoginFromAny" fromState="*" toState="login">
        ...
    </s:Transition>

    <!-- Play for a change to the login view state from the details view state. -->
    <s:Transition id="toLoginFromDetails" fromState="details" toState="login">
        ...
    </s:Transition>

    <!-- Play for a change from any view state to any other view state. -->
    <s:Transition id="toAnyFromAny" fromState="*" toState="*">
        ...
    </s:Transition>
</s:transitions>

<!-- Go to the login view state, transition toLoginFromAny plays. -->
<s:Button click="currentState="login";/>

<!-- Go to the details view state, transition toAnyFromAny plays. -->
<s:Button click="currentState="details";/>

<!-- Go to the login view state, transition toLoginFromDetails plays because you transitioned
from the details to the login view state. -->
<s:Button click="currentState="login";/>

<!-- Go to the default view state, transition toAnyFromAny plays. -->
<s:Button click="currentState='';/>
```

If a state change matches two transitions, the `toState` property takes precedence over the `fromState` property. If more than one transition matches, Flex uses the first matching definition detected in the `transition` Array.

Defining effect targets

The `<s:Transition>` tag shown in the section "Defining transitions" on page 1872 defines the effects that make up a transition. The top-level effect defines the target components of the effects in the transition when the effect does not explicitly define a target. In that example, the transition is performed on all three Panel containers in the application. If you want the transition to play only on the first two panels, you define the Parallel effects as the following example shows:

```
<s:Transition fromState="*" toState="*">
    <!-- Define a Parallel effect as the top most effect.-->
    <s:Parallel id="t1" targets="{[p1,p2]}">
        <s:Move duration="400"/>
        <s:Resize duration="400"/>
    </s:Parallel>
</s:Transition>
```

You removed the third panel from the transition, so it is no longer a target of the Move and Resize effects. Therefore, the third panel appears to jump to its new position and size during the change in view state. The other two panels show a smooth change in size and position for the 400-millisecond (ms) duration of the effects.

You can also use the `target` or `targets` properties of the effects within the transition to explicitly specify the effect target, as the following example shows:

```
<s:Parallel id="t1" targets="{[p1,p2,p3]}">
    <s:Move targets="{[p1,p2]}" duration="400"/>
    <s:Resize duration="400"/>
</s:Parallel>
```

In this example, the Resize effect plays on all three panels, while the Move effect plays only on the first two panels. You could also write this example as the following code shows:

```
<s:Parallel id="t1" targets="{[p1,p2]}">
    <s:Move duration="400"/>
    <s:Resize targets="{[p1,p2,p3]}" duration="400"/>
</s:Parallel>
```

Defining the effect start and end values

Like any effect, an effect within a transition has properties that you use to configure it. For example, most effects have properties that define starting and ending information for the target component, such as the `xFrom`, `yFrom`, `xTo`, and `yTo` properties of the Move effect.

Effects defined in a transition must determine their property values for the effect to execute. Flex uses the following rules to determine the start and end values of effect properties of a transition:

1 If the effect explicitly defines the values of any properties, use them in the transition, as the following example shows:

```
<s:Transition fromState="*" toState="*">
    <s:Sequence id="t1" targets="{[p1,p2,p3]}">
        <mx:Blur duration="100" blurXFrom="0.0" blurXTo="10.0"
            blurYFrom="0.0" blurYTo="10.0"/>
     <s:Parallel>
            <s:Move duration="400"/>
            <s:Resize duration="400"/>
        </s:Parallel>
     <mx:Blur duration="100" blurXFrom="10.0" blurXTo="0.0"
        blurYFrom="10.0" blurYTo="0.0"/>
    </s:Sequence>
</s:Transition>
```

In this example, the two Blur filters explicitly define the properties of the effect.

2 If the effect does not explicitly define the start values of the effect, Flex determines them from the current settings of the target component, as defined by the current view state.

In the example in rule 1, notice that the Move and Resize effects do not define start values. Therefore, Flex determines them from the current size and position of the effect targets in the current view state.

3 If the effect does not explicitly define the end values of the effect, Flex determines them from the settings of the target component in the destination view state.

In the example in rule 1, the Move and Resize effects determine the end values from the size and position of the effect targets in the destination view state. In some cases, the destination view state explicitly defines these values. If the destination view state does not define the values, Flex determines them from the settings of the default view state.

4 If there are no explicit values, and Flex cannot determine values from the current or destination view states, the effect uses its default property values.

The following example modifies the three-panel example shown in “Defining transitions” on page 1872 by adding Blur effects to the transition, where the Blur effect explicitly defines the values of the `blurXFrom`, `blurXTo`, `blurYFrom`, and `blurYTo` properties:

```
<?xml version="1.0" ?>
<!-- transitions\DefiningTransWithBlurs.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" width="400" >
    <!-- Define the two view states, in addition to the base state.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="One"/>
        <s:State name="Two"/>
    </s:states>

    <!-- Define the single transition for all view state changes.-->
    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence id="t1" targets="{[p1,p2,p3]}">
                <mx:Blur duration="100" blurXFrom="0.0" blurXTo="10.0"
                blurYFrom="0.0" blurYTo="10.0"/>
                <s:Parallel>
                    <mx:Move  duration="400"/>
                    <mx:Resize duration="400"/>
                </s:Parallel>
                <mx:Blur duration="100" blurXFrom="10.0" blurXTo="0.0"
                blurYFrom="10.0" blurYTo="0.0"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>

    <!-- Define the container holding the three Panel containers.-->
    <s:Group id="pm" width="100%" height="100%">
        <s:Panel id="p1" title="One"
                x="0" y="0"
                x.One="110" y.One="0"
                x.Two="0" y.Two="0"
                width="100" height="100"
                width.One="200" height.One="210"
                width.Two="100" height.Two="100"
                click="currentState='One'">
            <mx:Label fontSize="24" text="One"/>
        </s:Panel>

        <s:Panel id="p2" title="Two"
```

```
                x="0" y="110"
                x.One="0" y.One="0"
                x.Two="110" y.Two="0"
                width="100" height="100"
                width.One="100" height.One="100"
                width.Two="200" height.Two="210"
                click="currentState='Two'">
            <mx:Label fontSize="24" text="Two"/>
        </s:Panel>

        <s:Panel id="p3" title="Three"
                x="110" y="0"
                x.One="0" y.One="110"
                x.Two="0" y.Two="110"
                width="200" height="210"
                width.One="100" height.One="100"
                width.Two="100" height.Two="100"
                click="currentState='default'">
            <mx:Label fontSize="24" text="Three"/>
        </s:Panel>
    </s:Group>
</s:Application>
```

Interrupting and reversing a transition

Flex does not support the playing of multiple transitions simultaneously. If a transition is currently playing when a new transition occurs, by default the currently playing transition stops by calling the `end()` method on all effects in the transition.

For example, a transition is playing between view states A and B. While it is playing, a transition is triggered because of a change from view state B to view state C. The `end()` method causes the current transition, from view state A to B, to snap to the B view state. The new transition, from view state B to C, then starts playing.

Use the `Transition.interruptionBehavior` property of the new transition to control the behavior of the current transition. By default, the `interruptionBehavior` property is set to `end` to specify that the current transition snaps to its end view state. Set the `interruptionBehavior` property of the new transition to `stop` to halt that current transition instead. The new transition then plays from the halt location in the current transition, rather than having the current transition snap to its end view state. The value of `stop` can smooth out the appearance of an interrupted transition. That is because the user does not see the current transition snap to its end state before the new transition begins.

In some cases, the new transition is the reverse of the current transition. For example, a transition is playing between view states A and B. While it is playing, a transition is triggered to change back from view state B to A.

When your application requires a reverse transition, you can either:

- Defines two transitions. Define one transition for the change from view state A to view state B, and one for the change from B to A. Use the `toState` and `fromState` properties of the Transition class to specify the view states for the two transitions, as shown below:

```
<s:transitions>
    <s:Transition id="fromAtoB" fromState="A" toState="B">
        ...
    </s:Transition>

    <s:Transition id="fromBtoA" fromState="B" toState="A">
        ...
    </s:Transition>
</s:transitions>
```

The `interruptionBehavior` property controls the behavior of the current transition when the reverse transition starts playing.

- Define a single transition. Set the `autoReverse` property of the Transition class to `true` to specify that this transition applies to both the forward and reverse view state changes. Therefore, use this transition on a change from view state A to view state B, and on the change from B to A.

```
<s:transitions>
    <s:Transition id="fromAtoBtoA" fromState="A" toState="B" autoReverse="true">
        ...
</s:transitions>
```

***Note:** If a transition uses the `toState` and `fromState` properties to explicitly handle the transition from view state B to A, then Flex ignores the `autoReverse` property.*

While the transition from view state A to view state B is playing, the reverse transition can occur to interrupt the current transition. The reverse transition always halts the current transition at its current location. That is, the reverse transition always plays as if the `interruptionBehavior` property was set to `stop`, regardless of the real value of `interruptionBehavior`.

Handling events when using transitions

You can handle view state events, such as currentStateChange and currentStateChanging, as part of an application that defines transitions. Changes to the view state, dispatching events, and playing transition effects occur in the following order:

1 You set the `currentState` property to the destination view state.

2 Flex dispatches the `currentStateChanging` event.

3 Flex examines the list of transitions to determine the one that matches the change of the view state.

4 Flex examines the components to determine the start values of any effects.

5 Flex applies the destination view state to the application.

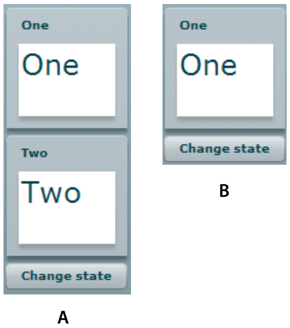
6 Flex dispatches the `currentStateChange` event.

7 Flex plays the effects that you defined in the transition.

If you change state again while a transition plays, Flex jumps to the end of the transition before starting any transition associated with the new change of view state.

Using action effects in a transition

In the following example, you define an application that has two view states:



A. *Default view state* ***B.*** *OneOnly view state*

To move from the default view state to the OneOnly view state, you create the following view state definition:

```
<s:states>
    <s:State name="default"/>
    <s:State name="OneOnly"/>
</s:states>
```

To move from the default view state to the OneOnly view state, you set the value of the `visible` and `includeInLayout` properties to `false` so that Flex makes the second Panel container invisible and ignores it when laying out the application. If the `visible` property is `false`, and the `includeInLayout` property is `true`, the container is invisible, but Flex lays out the application as if the component were visible.

A view state defines how to change states, and the transition defines the order in which the visual changes occur. In the example shown in the previous image, you play an Wipe effect on the second panel when it disappears, and when it reappears on a transition back to the base state.

For the change from the base state to the OneOnly state, you define the toOneOnly transition which uses the Wipe effect to make the second panel disappear, and then sets the panel's `visible` and `includeInLayout` properties to `false`. For a transition back to the base state, you define the toAnyFromAny transition that makes the second panel visible by setting its `visible` and `includeInLayout` properties to `true`, and then uses the Wipe effect to make the panel appear, as the following example shows:

```
<s:transitions>
    <s:Transition id="toOneOnly" fromState="*" toState="OneOnly">
        <s:Sequence id="t1" targets="{[p2]}">
            <s:Wipe direction="left" duration="350"/>
            <s:SetAction property="visible"/>
            <s:SetAction target="{p2}" property="includeInLayout"/>
        </Sequence>
    </s:Transition>

    <s:Transition id="toAnyFromAny" fromState="*" toState="*">
        <s:Sequence id="t2" targets="{[p2]}">
            <s:SetAction target="{p2}" property="includeInLayout"/>
            <s:SetAction property="visible"/>
            <s:Wipe direction="left" duration="350"/>
        </s:Sequence>
    </s:Transition>
</s:transitions>
```

In the toOneOnly transition, if you hide the target by setting its `visible` property to `false`, and then play the Iris effect, you would not see the Iris effect play because the target is already invisible.

To control the order of view state changes during a transition, Flex defines several *actioneffects*. The previous example uses the Spark SetAction action effect to control when to set the `visible` and `includeInLayout` properties in the transition. The following table describes the action effects:

Spark action effect	MX action effect	Corresponding view state class	Use
SetAction	SetPropertyAction	SetProperty	Sets a property value as part of a transition.
SetAction	SetStyleAction	SetStyle	Sets a style to a value as part of a transition.
AddAction	AddChildAction	AddChild	Adds a child as part of a transition.
RemoveAction	RemoveChildAction	RemoveChild	Removes a child as part of a transition.

Adobe recommends that you use the Spark action effects instead of the MX action effects.

To control when a change defined by the view state property occurs in a transition, you use the corresponding action effect. The action effects give you control over the order of the state change.

In the previous example, you used the following statement to define an action effect to occur when the value of the `visible` property of a component changes:

```
<s:SetAction property="visible"/>
```

This action effect plays when the value of the `visible` property changes to either `true` or `false`. You can further control the effect using the `value` property of the `<s:SetAction>` tag, as the following example shows:

```
<s:SetAction property="visible" value="true"/>
```

In this example, you specify to play the effect only when the value of the `visible` property changes to `true`. Adding this type of information to the action effect can be useful if you want to use filters with your transitions. For more information, see “Filtering effects” on page 1882.

The action effects do not support a `duration` property; they only perform the specified action.

Example: Using action effects

The following example shows the complete code for the example in “Using action effects in a transition” on page 1879:

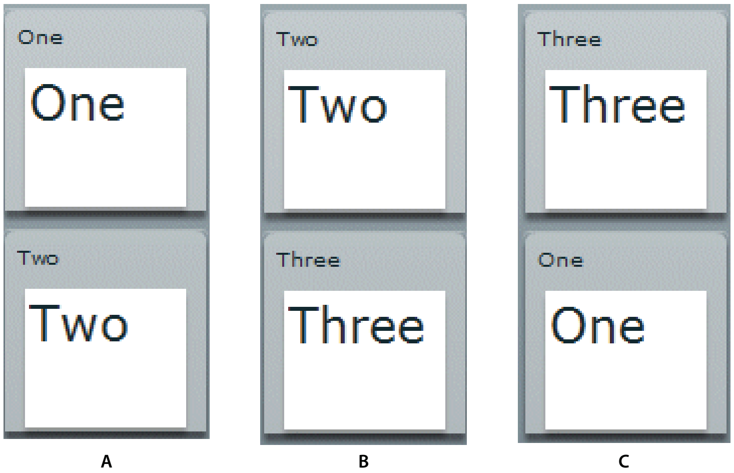
```
<?xml version="1.0" ?>
<!-- transitions\ActionTransitions.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Define one view state, in addition to the base state.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="OneOnly"/>
    </s:states>
    <!-- Define Transition array with one Transition object.-->
    <s:transitions>
        <s:Transition id="toOneOnly" fromState="*" toState="OneOnly">
            <s:Sequence id="t1" targets="{[p2]}">
                <s:Wipe direction="left" duration="350"/>
                <s:SetAction property="visible"/>
                <s:SetAction property="includeInLayout"/>
            </s:Sequence>
        </s:Transition>

        <s:Transition id="toAnyFromAny" fromState="*" toState="*">
            <s:Sequence id="t2" targets="{[p2]}">
                <s:SetAction property="includeInLayout"/>
                <s:SetAction property="visible"/>
                <s:Wipe direction="right" duration="350"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>
    <s:Panel id="p1" title="One"
        width="100" height="100">
        <s:Label fontSize="24" text="One"/>
    </s:Panel>

    <s:Panel id="p2" title="Two"
        width="100" height="100"
        visible="true" visible.OneOnly="false"
        includeInLayout="true" includeInLayout.OneOnly="false">
        <s:Label fontSize="24" text="Two"/>
    </s:Panel>

    <s:Button id="b1" label="Change state"
        click="currentState = currentState == 'OneOnly' ? '' : 'OneOnly';"/>
</s:Application>
```

Filtering effects

By default, Flex applies all of the effects defined in a transition to all of the target components of the transition. Therefore, in the following example, Flex applies the Move and Resize effects to all three targets:

```
<s:Transition fromState="*" toState="*">
    <!-- Define a Parallel effect as the top most effect.-->
    <s:Parallel id="t1" targets="{[p1,p2,p3]}">
        <!-- Define a Move and Resize effect.-->
        <s:Move duration="400"/>
        <s:Resize duration="400"/>
    </s:Parallel>
</s:Transition>
```

However, you might want to conditionalize an effect so that it does not apply to all target components, but only to a subset of the components. For example, you define an application with three view states, as the following image shows:



A. *Default view state* ***B.*** *Two view state* ***C.*** *Three view state*

Each change of view state removes the top panel, moves the bottom panel to the top, and adds the next panel to the bottom of the screen. In this example, the third panel is invisible in the default view state.

For this example, you define a single transition that applies a Wipe effect to the top panel as it is removed, applies a Move effect to the bottom panel as it moves to the top position, and applies another Wipe effect to the panel being added to the bottom, as the following example shows:

```
<s:transitions>
    <s:Transition fromState="*" toState="*">
        <s:Sequence targets="{[p1,p2,p3]}">
            <s:Sequence id="sequence1" filter="hide">
                <s:Wipe direction="up"/>
                <s:SetPropertyAction name="visible" value="false"/>
            </s:Sequence>
            <s:Move filter="move"/>
            <s:Sequence id="sequence2" filter="show">
                <s:SetAction property="visible" value="true"/>
                <s:Wipe direction="up"/>
            </s:Sequence>
        </s:Sequence>
    </s:Transition>
</s:transitions>
```

The sequence1 Sequence effect uses the `filter` property to specify the change that a component must go through in order for the effect to play on it. In this example, the sequence1 effect specifies a value of `"hide"` for the `filter` property. Therefore, the Wipe and SetAction effects only play on those components that change from visible to invisible by setting their `visible` property to `false`.

In the sequence2 Sequence effect, you set the `filter` property to `show`. Therefore, the Wipe and SetAction effects only play on those components whose state changes from invisible to visible by setting their `visible` property to `true`.

The Move effect also specifies a `filter` property with a value of `move`. Therefore, it applies to all target components that are changing position.

The following table describes the possible values of the `filter` property:

Value	Description
`add`	Specifies to play the effect on all children added during the change of view state.
`hide`	Specifies to play the effect on all children whose `visible` property changes from `true` to `false` during the change of view state.
`move`	Specifies to play the effect on all children whose `x` or `y` properties change during the change of view state.
`remove`	Specifies to play the effect on all children removed during the change of view state.
`resize`	Specifies to play the effect on all children whose `width` or `height` properties change during the change of view state.
`show`	Specifies to play the effect on all children whose `visible` property changes from `false` to `true` during the change of view state.

Example: Using a filter

The following example shows the complete code for the example in “Filtering effects” on page 1882:

```
<?xml version="1.0" ?>
<!-- transitions\FilterShowHide.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark" width="700" >
    <!-- Define two view state, in addition to the base state.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="Two"/>
        <s:State name="Three"/>
    </s:states>

    <!-- Define a single transition for all state changes.-->
    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence targets="{[p1,p2,p3]}">
                <s:Sequence id="sequence1" filter="hide" >
                    <s:Wipe direction="up"/>
                    <s:SetAction property="visible" value="false"/>
                </s:Sequence>
                <s:Move filter="move"/>
                <s:Sequence id="sequence2" filter="show" >
                    <s:SetAction property="visible" value="true"/>
                    <s:Wipe direction="up"/>
                </s:Sequence>
            </s:Sequence>
        </s:Transition>
    </s:transitions>
    <s:Group id="pm" width="100%" height="100%">
        <s:Panel id="p1" title="One"
                visible="true" visible.Two="false"
                x="0" y="0"
                x.Three="0" y.Three="110"
                width="100" height="100"
                click="currentState=''" >
            <s:Label fontSize="24" text="One"/>
        </s:Panel>
        <s:Panel id="p2" title="Two"
                visible="true" visible.Three="false"
                x="0" y="110"
                x.Two="0" y.Two="0"
                width="100" height="100"
                click="currentState='Two'" >
            <s:Label fontSize="24" text="Two"/>
        </s:Panel>

        <s:Panel id="p3" title="Three"
                visible="false" visible.Two="true" visible.Three="true"
                x.Two="0" y.Two="110"
                x.Three="0" y.Three="0"
                width="100" height="100"
                click="currentState='Three'" >
            <s:Label fontSize="24" text="Three"/>
        </s:Panel>
    </s:Group>
</s:Application>
```

Defining a custom filter

Flex lets you use the EffectTargetFilter class to define a custom filter that is executed by each transition effect on each target of the effect. By defining a custom filter, you can specify your own logic for controlling a transition. The following table describes the properties of the EffectTargetFilter class:

Property	Description
`filterProperties`	An Array of Strings specifying component properties. If any of the properties in the Array have changed on the target component, play the effect on the target.
`filterStyles`	An Array of Strings specifying style properties. If any of the style properties in the Array have changed on the target component, play the effect on the target.
`filterFunction`	A property containing a reference to a callback function that defines custom filter logic. Flex calls this method on every target of the effect. If the function returns `true`, the effect plays on the target; if it returns `false`, the target is skipped by that effect.
`requiredSemantics`	A collection of properties and associated values which must be associated with a target for the effect to be played.

The callback function specified by the `filterFunction` property has the following signature:

```
filterFunc(propChanges:Array, instanceTarget:Object):Boolean {
    // Return true to play the effect on instanceTarget,
    // or false to not play the effect.
}
```

The function takes the following arguments:

propChanges An Array of PropertyChanges objects, one object per target component of the effect. If a property of a target is not modified by the transition, it is not included in this Array.

instanceTarget The specific target component of the effect that you filter.

For an example using a custom filter function, see “Example: Using a custom effect filter” on page 1889.

The following example defines a custom filter that specifies to play the effect only on a target component whose `x` or `width` property is modified as part of the change of view state:

```
<?xml version="1.0" ?>
<!-- transitions\EffectFilterExampleMXML.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="700">

    <!-- Define the two view states, in addition to the base state.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="One"/>
        <s:State name="Two"/>
    </s:states>

    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence id="t1" targets="{[p1,p2,p3]}">
                <mx:Blur id="myBlur" duration="100" blurXFrom="0.0"
                    blurXTo="10.0" blurYFrom="0.0" blurYTo="10.0">
                    <!-- Define the custom filter. -->
                    <mx:customFilter>
                        <mx:EffectTargetFilter
                            filterProperties="['width','x']"/>
                    </mx:customFilter>
                </mx:Blur>
                <s:Parallel>
                    <s:Move  duration="400"/>
                    <s:Resize duration="400"/>
                </s:Parallel>
                <mx:Blur id="myUnBlur" duration="100" blurXFrom="10.0"
                    blurXTo="0.0" blurYFrom="10.0" blurYTo="0.0">
                    <!-- Define the custom filter. -->
                    <mx:customFilter>
                        <mx:EffectTargetFilter
                            filterProperties="['width','x']"/>
                    </mx:customFilter>
                </mx:Blur>
            </s:Sequence>
        </s:Transition>
    </s:transitions>

    <!-- Define the container holding the three Panel containers.-->
    <s:Group id="pm" width="100%" height="100%">
        <s:Panel id="p1" title="One"
                x="0" y="0"
                x.One="110" y.One="0"
                x.Two="0" y.Two="0"
                width="100" height="100"
                width.One="200" height.One="210"
                width.Two="100" height.Two="100"
                click="currentState='One'">
            <s:Label fontSize="24" text="One"/>
        </s:Panel>

        <s:Panel id="p2" title="Two"
                x="0" y="110"
                x.One="0" y.One="0"
                x.Two="110" y.Two="0"
                width="100" height="100"
                width.One="100" height.One="100"
                width.Two="200" height.Two="210"
                click="currentState='Two'">
            <s:Label fontSize="24" text="Two"/>
        </s:Panel>

        <s:Panel id="p3" title="Three"
                x="110" y="0"
                x.One="0" y.One="110"
                x.Two="0" y.Two="110"
                width="200" height="210"
                width.One="100" height.One="100"
                width.Two="100" height.Two="100"
                click="currentState='default'">
            <s:Label fontSize="24" text="Three"/>
        </s:Panel>
    </s:Group>
</s:Application>
```

Writing a filter function

To create a filter function, you define an EffectTargetFilter object, and then specify that object as the value of the `Effect.customFilter` property for an effect. The following example uses the `creationComplete` event of an application to initialize an EffectTargetFilter object, and then specify it as the value of the `customFilter` property for two Blur effects:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initFilter(event);" width="700">

    <fx:Script>
        <![CDATA[

            import mx.effects.EffectTargetFilter;
            import flash.events.Event;

            // This function returns true for the Panel moving to x=110.
            public function filterFunc(propChanges:Array,instanceTarget:Object):Boolean
            {
                ...
            }

            // Define the EffectTargetFilter object.
            private var myBlurFilter:EffectTargetFilter;

            // Initialize the EffectTargetFilter object, and set the
            // customFilter property for two Blur effects.
            private function initFilter(event:Event):void {
                myBlurFilter = new EffectTargetFilter();

                myBlurFilter.filterFunction=filterFunc;

                myBlur.customFilter=myBlurFilter;
                myUnBlur.customFilter=myBlurFilter;
            }

        ]]>
    </Script>

    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence id="t1" targets="{[p1,p2,p3]}">
                <mx:Blur id="myBlur"/>
                    <s:Parallel>
                        <s:Move duration="400"/>
                        <s:Resize duration="400"/>
                    </Parallel>
                <mx:Blur id="myUnBlur"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>

    ...

</s:Application>
```

The `propChanges` argument passed to the filter function contains an Array of PropertyChanges objects, one object per target component of the effect. The following table describes the properties of the PropertyChanges class:

Property	Description
`target`	A target component of the effect. The `end` and `start` properties of the PropertyChanges class define how the target component is modified by the change to the view state.
`start`	An Object that contains the starting properties of the `target` component, as defined by the current view state. For example, for a `target` component that is both moved and resized by a change to the view state, the `start` property contains the starting position and size of the component, as the following example shows: `{x:00, y:00, width:100, height:100}`
`end`	An Object that contains the ending properties of the `target` component, as defined by the destination view state. For example, for a `target` component that is both moved and resized by a change to the view state, the `end` property contains the ending position and size of the component, as the following example shows: `{x:100, y:100, width:200, height:200}`

Within the custom filter function, you first search the `propChanges` Array for the PropertyChanges object that matches the `instanceTarget` argument by comparing the `instanceTarget` argument to the `propChanges.target` property.

The following filter function examines the `propChanges` Array to determine if it should play the effect on the `instanceTarget`. In this example, the filter function returns `true` only for those components being moved to a position where the `translationX` property equals 110, as the following example shows:

```
// This function returns true for a target moving to x=110.
public function filterFunc(propChanges:Array, instanceTarget:Object):Boolean {

    // Determine the length of the propChanges Array.
    for (var i:uint=0; i < propChanges.length; i++)
    {
        // Determine the Array element that matches the effect target.
        if (propChanges[i].target == instanceTarget)
        {
            // Check to see if the end Object contains a value for x.
            if (propChanges[i].end["translationX"] != undefined)
            {
                // Return true of the end value for x is 110.
                return (propChanges[i].end.translationX == 110);
            }
        }
    }
    // Otherwise, return false.
    return false;
}
```

Example: Using a custom effect filter

In the following example, you use a custom filter function to apply Blur effects to one of the three targets of a transition. The other two targets are not modified by the Blur effects.

To determine the target of the Blur effects, the custom filter function examines the `translationX` property of each target. The Blur effects play only on the component moving to x=110, as the following example shows:

```
<?xml version="1.0" ?>
<!-- transitions\EffectFilterExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initFilter(event);"
    width="700">

    <fx:Script>
        <![CDATA[
            import mx.effects.EffectTargetFilter;
            import flash.events.Event;
            // This function returns true for the Panel moving to x=110.
            public function filterFunc(propChanges:Array,
                    instanceTarget:Object):Boolean {
                // Determine the length of the propChanges Array.
                for (var i:uint=0; i < propChanges.length; i++)
                {
                    // Determine the Array element
                    // that matches the effect target.
                    if (propChanges[i].target == instanceTarget)
                    {
                        // Check whether the end Object contains
                        // a value for x.
                        if (propChanges[i].end["translationX"] != undefined)
                        {
                            // Return true if the end value for x is 110.
                            return (propChanges[i].end.translationX == 110);
                        }
                    }
                }
                return false;
            }
            // Define the EffectTargetFilter object.
            private var myBlurFilter:EffectTargetFilter;
            // Initialize the EffectTargetFilter object, and set the
            // customFilter property for two Blur effects.
            private function initFilter(event:Event):void {
                myBlurFilter = new EffectTargetFilter();
                myBlurFilter.filterFunction=filterFunc;
                myBlur.customFilter=myBlurFilter;
                myUnBlur.customFilter=myBlurFilter;
            }
          ]]>
    </fx:Script>
    <!-- Define the two view states, in addition to the base state.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="One"/>
        <s:State name="Two"/>
    </s:states>

    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence id="t1" targets="{[p1,p2,p3]}">
                <mx:Blur id="myBlur" duration="100" blurXFrom="0.0"
                    blurXTo="10.0" blurYFrom="0.0" blurYTo="10.0"/>
                <s:Parallel>
                    <s:Move  duration="400"/>
                    <s:Resize duration="400"/>
                </s:Parallel>
                <mx:Blur id="myUnBlur" duration="100" blurXFrom="10.0"
                    blurXTo="0.0" blurYFrom="10.0" blurYTo="0.0"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>
    <!-- Define the container holding the three Panel containers.-->
    <s:Group id="pm" width="100%" height="100%">
        <s:Panel id="p1" title="One"
                x="0" y="0"
                x.One="110" y.One="0"
                x.Two="0" y.Two="0"
                width="100" height="100"
                width.One="200" height.One="210"
                width.Two="100" height.Two="100"
                click="currentState='One'">
            <s:Label fontSize="24" text="One"/>
        </s:Panel>

        <s:Panel id="p2" title="Two"
                x="0" y="110"
                x.One="0" y.One="0"
```

```
                x.Two="110" y.Two="0"
                width="100" height="100"
                width.One="100" height.One="100"
                width.Two="200" height.Two="210"
                click="currentState='Two'">
            <s:Label fontSize="24" text="Two"/>
        </s:Panel>

        <s:Panel id="p3" title="Three"
                x="110" y="0"
                x.One="0" y.One="110"
                x.Two="0" y.Two="110"
                width="200" height="210"
                width.One="100" height.One="100"
                width.Two="100" height.Two="100"
                click="currentState='default'">
            <s:Label fontSize="24" text="Three"/>
        </s:Panel>
    </s:Group>
</s:Application>
```

Using the requiredSemantics property

When working with data effects, you can use the `EffectTargetFiler.requiredSemantics` property to filter effects. If you want to play a data effect on all targets of a list control that are not added by the effect, meaning targets that are removed, replaced, moved, or affected in any other way, you can write the effect definition as shown below:

```
<mx:Blur>
    <mx:customFilter>
        <mx:EffectTargetFilter requiredSemantics="{{'added':false}}"/>
    </mx:customFilter>
</mx:Blur>
```

To play a data effect on all targets that are not added or not removed by the effect, you can write the effect definition as shown below:

```
<mx:Blur>
    <mx:customFilter>
        <mx:EffectTargetFilter requiredSemantics="{{'added':false}, {'removed':false}}"/>
    </mx:customFilter>
</mx:Blur>
```

The allowed list of properties that you can specify includes `added`, `removed`, `replaced`, and `replacement`. The allowed values for the properties are `true` and `false`.

For more information on data effects, see “Using MX data effects” on page 1839.

Transition tips and troubleshooting

Tips

Determine the type of transition you are defining:

- With *dynamic* transitions, you know what effects you want to play, but not which targets they will play on.
- With *explicit* transitions, you know exactly what happens to each individual target.

Complex transitions may consist of both dynamic and explicit elements.

Tips for dynamic transitions:

- List all possible targets in the parent composite effect.
- By default, all effects play on all specified targets. Use filtering with dynamic transitions to limit the target set.
- Dynamic transitions can be used with a wide variety of state changes.

Tips for explicit transitions:

- Specify targets on the child effects, or on a composite effect when all child effects of the composite effect have the same targets.
- By default, all effects play on all specified targets. For explicit transitions, make sure the targets are correctly set.
- Explicit transitions typically require a different transition for each change to the view state.

Troubleshooting

Troubleshooting a transition effect that does not play:

- Is the effect target being hidden or removed? If so, make sure you add an `<RemoveChild>` property to the view state definition, or an `<s:SetPropertyAction name="visible">` tag in the transition definition.
- Does the change to the view state define settings that pertain to the transition effect? For example, if you have a Resize effect, you must change the `width` or `height` property of the target when the view state changes to trigger the effect.
- Check that you specified the correct targets to the transition.
- Check that your filter settings on the effect and on any parent effect are not excluding the target.

Troubleshooting an effect playing on too many targets:

- Add a filter for a dynamic transition, or change the targets for an explicit transition.

Troubleshooting wrong effect parameters:

- Did you specify explicit parameters on the effect? Are they correct?
- Ensure that you correctly set the `showTarget` property for mask effects such as the Iris effect, and the wipe effects.

Troubleshooting a flickering or flashing target:

- Ensure that you correctly set the `showTarget` property for mask effects such as the Iris effect, and the wipe effects.
- Is the effect target being hidden or removed? If so, make sure you add an `<RemoveChild>` property to the view state definition to remove the target, or an `<SetPropertyAction name="visible">` tag in the transition definition to hide the target.

Troubleshooting when an application does not look correct after a transition:

- Some effects leave the target with changed properties. For example, a Fade effect leaves the `alpha` property of the target at the `alphaTo` value specified to the effect. If you use the Fade effect on a target that sets `alpha` property to zero, you must set the `alpha` property to a nonzero value before the object appears again.
- Try removing one effect at a time from the transition until it no longer leaves your application with the incorrect appearance.

Drag and drop

The drag-and-drop operation lets you move data from one place in an application to another. It is especially useful in a visual application where you can drag data between two lists, drag controls in a container to reposition them, or drag Flex components between containers.

About drag and drop

Visual development environments typically let you manipulate objects in an application by selecting them with a mouse and moving them around the screen. Drag and drop lets you select an object, such as an item in a List control, or a Flex control such as an Image control, and then drag it over another component to add it to that component.

You can add support for drag and drop to all Flex components. Flex also includes built-in support for the drag-and-drop operation for certain controls such as the MX List, Tree, and DataGrid controls and the Spark List control, that automate much of the processing required to support drag and drop.

About the drag-and-drop operation

The drag-and-drop operation has three main stages: initiation, dragging, and dropping:

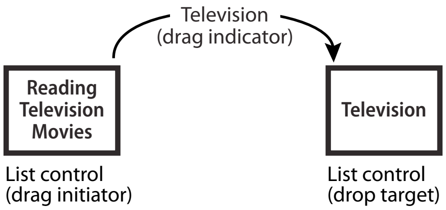
Initiation User initiates a drag-and-drop operation by using the mouse to select a Flex component, or an item in a Flex component, and then moving the component or item while holding down the mouse button. For example, a user selects an item in a List control with the mouse and, while holding down the mouse button, moves the mouse several pixels. The selected component, the List control in this example, is the *drag initiator*.

Dragging While still holding down the mouse button, the user moves the mouse around the application. Flex displays an image during the drag, called the *drag indicator*. A *drag source* object (an object of type DragSource) contains the data being dragged.

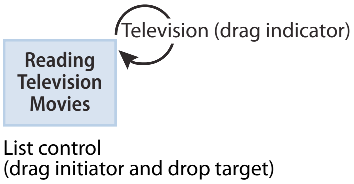
Dropping When the user moves the drag indicator over another Flex component, that component becomes a possible *drop target*. The drop target inspects the drag source object to determine whether the data is in a format that the target accepts and, if so, allows the user to drop the data onto it. If the drop target determines that the data is not in an acceptable format, the drop target disallows the drop.

A drag-and-drop operation either copies or moves data from the drag initiator to the drop target. Upon a successful drop, Flex adds the data to the drop target and, optionally, deletes it from the drag initiator in the case of a move.

The following figure shows one List control functioning as the drag initiator and a second List control functioning as the drop target. In this example, you use drag and drop to move the ’Television’ list item from the drag initiator to the drop target:



A single component can function as both the drag initiator and the drop target. This lets you move the data within the component. The following example shows a List control functioning as both the drag initiator and the drop target:



By specifying the List control as both the drag initiator and the drop target, you can use drag and drop to rearrange the items in the control. For example, if you use a Canvas container as the drag initiator and the drop target, you can then use drag and drop to move controls in the Canvas container to rearrange them.

Performing a drag and drop

Drag and drop is event driven. To configure a component as a drag initiator or as a drop target, you have to write event handlers for specific events, such as the `dragDrop` and `dragEnter` events. For more information, see "Manually adding drag-and-drop support" on page 1902.

For some components that you often use with drag and drop, Flex provides built-in event handlers to automate much of the drag and drop operation. These controls are all subclasses of the MX mx.controls.listClasses.ListBase class and the Spark spark.components.supportClasses.ListBase class, and are referred to as list-based controls. For more information, see "Using drag-and-drop with list-based controls" on page 1894.

For a move or copy operation, the list-based controls can handle all of the events required by a drag-and-drop operation. However, if you want to copy the drag data to the drop target, and the drop target uses a different data format, you have to write an event handler. For more information, see "Moving and copying data" on page 1927.

Using drag-and-drop with list-based controls

The following MX list-based controls include built-in support for the drag-and-drop operation:

- DataGrid
- HorizontalList
- List
- PrintDataGrid
- TileList
- Tree

The Spark List control include built-in support for the drag-and-drop operation.

The built-in support for these controls lets you move and copy items by dragging them from a drag-enabled control to a drop-enabled control.

The following drag-and-drop example lets you move items from one Spark List control to another:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToListMove.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            private function initApp():void {
                srclist.dataProvider =
                    new ArrayCollection(['Reading', 'Television', 'Movies']);
                destlist.dataProvider = new ArrayCollection([]);
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:VGroup>
            <s:Label text="Available Activities"/>
            <s:List id="srclist"
                allowMultipleSelection="true"
                dragEnabled="true"
                dragMoveEnabled="true"/>
        </s:VGroup>
        <s:VGroup>
            <s:Label text="Activities I Like"/>
            <s:List id="destlist"
                dropEnabled="true"/>
        </s:VGroup>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

By setting the `dragEnabled` property to `true` on the first List and the `dropEnabled` property to `true` on the second List control, you enabled users to drag items from the first list to the second without worrying about any of the underlying event processing.

For all list-based classes except the Tree control, the default value of the `dragMoveEnabled` property is `false`, so you can only copy elements from one control to the other. By setting the `dragMoveEnabled` to `true` in the first List control, you can move and copy data. For the Tree control, the default value of the `dragMoveEnabled` property is `true`.

When the `dragMoveEnabled` property is set to `true`, the default drag-and-drop action is to move the drag data. To perform a copy, hold down the Control key during the drag-and-drop operation.

The only requirement on the drag and drop operation is that the structure of the data providers must match for the two controls. In this example, the data provider for srclist is an Array of Strings, and the data provider for the destination List control is an empty Array. If the data provider for destlist is an Array of some other type of data, destlist might not display the dragged data correctly.

You can modify the dragged data as part of a drag-and-drop operation to make the dragged data compatible with the destination. For an example of dragging data from one control to another when the data formats do not match, see "Example: Copying data from an MX List control to an MX DataGrid control" on page 1928.

You can allow two-way drag and drop by setting the `dragEnabled`, `dropEnabled`, and `dragMoveEnabled` properties to `true` for both list-based controls, as the following example shows for two MX DataGrid controls. In this example, you can drag and drop rows from either MX DataGrid control to the other:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleDGToDG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
      <![CDATA[
          import mx.collections.ArrayCollection;

          private function initApp():void {
            srcgrid.dataProvider = new ArrayCollection([
              {Artist:'Carole King', Album:'Tapestry', Price:11.99},
              {Artist:'Paul Simon', Album:'Graceland', Price:10.99},
              {Artist:'Original Cast', Album:'Camelot', Price:12.99},
              {Artist:'The Beatles', Album:'The White Album', Price:11.99}
            ]);
            destgrid.dataProvider = new ArrayCollection([]);
          }
      ]]>
    </fx:Script>
    <s:HGroup>
        <s:VGroup>
            <s:Label text="Available Albums"/>
            <mx:DataGrid id="srcgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true">
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"/>
                    <mx:DataGridColumn dataField="Album"/>
                    <mx:DataGridColumn dataField="Price"/>
                </mx:columns>
            </mx:DataGrid>
        </s:VGroup>
        <s:VGroup>
            <s:Label text="Buy These Albums"/>
            <mx:DataGrid id="destgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true">
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"/>
                    <mx:DataGridColumn dataField="Album"/>
                    <mx:DataGridColumn dataField="Price"/>
                </mx:columns>
            </mx:DataGrid>
        </s:VGroup>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp()"
    />
</s:Application>
```

Dragging and dropping between different list-based controls

You can drag and drop data between different types of list-based controls. For example, you can drag and drop between an MX List and a Spark List control, as the following example shows:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToListMoveSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            private function initApp():void {
                srclist.dataProvider =
                    new ArrayList(['Reading', 'Television', 'Movies']);
                destlist.dataProvider = new ArrayList([]);
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:VGroup>
            <s:Label text="Available Activities"/>
            <s:List id="srclist"
                allowMultipleSelection="true"
                dragEnabled="true"
                dragMoveEnabled="true"/>
        </s:VGroup>
        <s:VGroup>
            <s:Label text="Activities I Like"/>
            <mx:List id="destlist"
                dropEnabled="true"/>
        </s:VGroup>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

Dragging and dropping between two types of List controls is relatively simple because the control use the same data format. However, when you drag and drop between a List and a DataGrid, or a List and a Tree, the data formats might not match. In that case, you have to manipulate the data before dropping it onto the target. see “Example: Copying data from an MX List control to an MX DataGrid control” on page 1928.

Dragging and dropping in the same control

One use of drag and drop is to let you reorganize the items in a list-based control by dragging the items and then dropping them in the same control. In the next example, you define an MX Tree control, and let the user reorganize the nodes of the Tree control by dragging and dropping them. In this example, you set the `dragEnabled` and `dropEnabled` to `true` for the Tree control (the `dragMoveEnabled` property defaults to `true` for the Tree control):

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleTreeSelf.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            // Initialize the data provider for the Tree.
            private function initApp():void {
                firstList.dataProvider = treeDP;
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:XML id="treeDP">
            <node label="Mail">
                <node label="Inbox"/>
                <node label="Personal Folder">
                    <node label="Demo"/>
                    <node label="Personal"/>
                    <node label="Saved Mail"/>
                    <node label="bar"/>
                </node>
                <node label="Calendar"/>
                <node label="Sent"/>
                <node label="Trash"/>
            </node>
        </fx:XML>
    </fx:Declarations>
    <mx:Tree id="firstList"
        showRoot="false"
        labelField="@label"
        dragEnabled="true"
        dropEnabled="true"
        allowMultipleSelection="true"
        creationComplete="initApp();"/>
</s:Application>
```

Drag and drop properties for list-based controls

List-based controls provide properties and methods for managing the drag-and-drop operation. The following table lists these properties and methods:

Property/Method	Description
`dragEnabled`	A Boolean value that specifies whether the control is a drag initiator. The default value is `false`. When `true`, users can drag selected items from the control. When a user drags items from the control, Flex creates a DragSource object that contains the following data objects: • The drag data as a Vector of type Object. Each element of the Vector corresponds to a dragged item from the data provider of the drag initiator. If you drag a single item, the Vector has a length of 1. For all controls except for Tree, the format string of the drag data is `"itemsByIndex"`, and the items implement the IDataProvider interface. • For Tree controls the drag data is an Array. The format string of the drag data is "`treeItems`" and the items implement the ITreeDataProvider API interface. • The index of the item that was clicked by the mouse. The format string of the index is “`caretIndex`”. The index is relative to the items in the drag data. Therefore, if the drag data contains three items, the index is a value between 0 and 2. By default, only the Spark list-based controls working as a drop target use this information to update the selected item on the drop. • A reference to the drag initiator, with a format String of `"source"` .
`dropEnabled`	A Boolean value that specifies whether the control can be a drop target. The default value is `false`, which means that you must write event handlers for the drag events. When the value is `true`, you can drop items onto the control by using the default drop behavior.
`dragMoveEnabled`	If the value is `true`, and the `dragEnabled` property is `true`, specifies that you can move or copy items from the drag initiator to the drop target. When you move an item, the item is deleted from the drag initiator when you add it to the drop target. If the value is `false`, you can only copy an item to the drop target. For a copy, the item in the drag initiator is not affected by the drop. When the `dragMoveEnabled` property is `true`, you must hold down the Control key during the drop operation to perform a copy. The default value is `false` for all list controls except the Tree control, and `true` for the Tree control.

Maintaining type information during a copy

When you use the built-in support of the list-based controls to copy data from one list-based control to another list-based control, you might run into an occasion where you lose the data-type information of the copied data. The loss of data-type information can occur when:

- You perform a copy operation between two list-based controls; it does not occur during a move operation
- The data type of the copied item is not a basic ActionScript data type, such as Date, Number, Object, or String
- The data type of the copied item is not DisplayObject, or a subclass of DisplayObject

For example, you define the following class, Car.as, that you use to populate the data provider of a List control:

```
package
{
    // dragdrop/Car.as

    [RemoteClass]
    public class Car extends Object
    {
        // Constructor.
        public function Car()
        {
            super();
        }

        // Class properties.
        public var numWheels:int;
        public var model:String;
        public var make:String;

        public function get label():String
        {
            return make + " " + model;
        }
    }
}
```

Notice that the Car.as file includes the `[RemoteClass]` metadata tag. This metadata tag is required to register the Car data type with Flex so that its type information is preserved during the copy operation. If you omit the `[RemoteClass]` metadata tag, type information is lost.

You then use that class in your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDRemoteClassListUpdated.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns="*">
    <s:layout>
        <s:BasicLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            public function changeit():void
            {
                if (list2.dataProvider != null)
                {
                    msg.text += list2.dataProvider[0]

                    if(list2.dataProvider[0] is Car)
                        msg.text += " Is Car\n";
                    else
                        msg.text += " Is NOT Car\n";
                }
            }
        ]]>
    </fx:Script>
    <s:List id="list1"
        x="10" y="45"
        width="160" height="120"
        dragEnabled="true"
        dragMoveEnabled="true">
        <s:dataProvider>
            <s:ArrayCollection>
                <Car model="Camry" make="Toyota" numWheels="4"/>
                <Car model="Prius" make="Toyota" numWheels="4"/>
            </s:ArrayCollection>
       </s:dataProvider>
    </s:List>
    <s:List  id="list2"
        x="200" y="45"
        width="160" height="120"
        dropEnabled="true"/>

    <s:Button label="Access it as button" click="changeit();"/>

    <s:TextArea id="msg"
        x="10" y="200"
        width="400" height="100"/>
</s:Application>
```

Manually adding drag-and-drop support

The list-based controls have built-in support for drag and drop, but you can use drag and drop with any Flex component. To support drag-and-drop operations with components other than MX list-based controls, or to explicitly control drag and drop with MX list-based controls, you must handle the drag and drop events.

Classes used in drag-and-drop operations

You use the following classes to implement the drag-and-drop operation:

Class	Function
DragManager	Manages the drag-and-drop operations; for example, its `doDrag()` method starts the drag operation.
DragSource	Contains the data being dragged. It also provides additional drag management features, such as the ability to add a handler that is called when data is requested.
DragEvent	Represents the event object for all drag-and-drop events.

Drag-and-drop events for a drag initiator

A component that acts as a drag initiator handles the following events to manage the drag-and-drop operation:

Drag initiator event	**Description**	**Handler required**	**Implemented by list-based controls**
mouseDown and mouseMove	The mouseDown event is dispatched when the user selects a control with the mouse and holds down the mouse button. The mouseMove event is dispatched when the mouse moves. For most controls, you initiate the drag-and-drop operation in response to one of these events. For an example, see "Example: Handling the dragOver and dragExit events for the drop target" on page 1924.	Yes, for nonlist controls	No
dragStart	Dispatched by a list-based component when the user initiates a drag operation. This event is used internally by the list-based controls; you do not handle it when implementing drag and drop. If you want to control the start of a drag-and-drop operation, use the mouseDown or mouseMove event.	Yes, for list controls	Yes
dragComplete	Dispatched when a drag operation completes, either when the drag data drops onto a drop target, or when the drag-and-drop operation ends without performing a drop operation. You can use this event to perform any final cleanup of the drag-and-drop operation. For example, if a user moves data from one component to another, you can use this event to delete the item from the drag initiator. For an example, see "Example: Moving and copying data for a nonlist-based control" on page 1929.	No	Yes

When adding drag-and-drop support to a component, you must implement an event handler for either the mouseDown or mouseMove event, and optionally for the dragComplete event. When you set the dragEnabled property to true for a list-based control, Flex automatically adds event handlers for the dragStart and dragComplete events.

Note: *Do not add an event handler for the* dragStart *event. That is an internal event handled by Flex.*

Drag-and-drop events for a drop target

To use a component as a drop target, you handle the following events:

Drop target event	Description	Handler required	Implemented by list-based controls
`dragEnter`	Dispatched when a drag indicator moves over the drop target from outside the drop target. A component *must* define an event handler for this event to be a drop target. The event handler determines whether the data being dragged is in an accepted format. To accept the drop, the event handler calls the `DragManager.acceptDragDrop()` method. You must call the `DragManager.acceptDragDrop()` method for the drop target to receive the `dragOver`, `dragExit`, and `dragDrop` events. In the handler, you can change the appearance of the drop target to provide visual feedback to the user that the component can accept the drag operation. For example, you can draw a border around the drop target, or give focus to the drop target. For an example, see "Example: Simple drag-and-drop operation for a nonlist-based control" on page 1906.	Yes	Yes
`dragOver`	Dispatched while the user moves the mouse over the target, after the `dragEnter` event. You can handle this event to perform additional logic before allowing the drop operation, such as adding the data to various locations within the drop target, reading keyboard input to determine if the drag-and-drop operation is a move or copy of the drag data, or providing different types of visual feedback based on the type of drag-and-drop operation. For an example, see "Example: Handling the dragOver and dragExit events for the drop target" on page 1924.	No	Yes
`dragDrop`	Dispatched when the user releases the mouse over the drop target. Use this event handler to add the drag data to the drop target. For an example, see "Example: Simple drag-and-drop operation for a nonlist-based control" on page 1906.	Yes	Yes
`dragExit`	Dispatched when the user moves the drag indicator off of the drop target, but does not drop the data onto the target. You can use this event to restore the drop target to its normal appearance if you modified its appearance in response to a `dragEnter` event or other event. For an example, see "Example: Handling the dragOver and dragExit events for the drop target" on page 1924.	No	Yes

When adding drag-and-drop support to a nonlist-based component, you must implement an event handler for the `dragEnter` and `dragDrop` events, and optionally for the other events. When you set the `dropEnabled` property to `true` for a list-based control, Flex automatically adds event handlers for all events.

The drag-and-drop operation

The following steps define the drag-and-drop operation.

1. A component becomes a drag-and-drop initiator in either of the following ways:
   - List-based components with `dragEnabled=true`

     Flex automatically makes the component an initiator when the user clicks and moves the mouse on the component.
   - Nonlist-based components, or list-based components with `dragEnabled=false`

     The component must detect the user's attempt to start a drag operation and explicitly become an initiator. Typically, you use the `mouseMove` or `mouseDown` event to start the drag-and-drop operation.

     The component then creates an instance of the mx.core.DragSource class that contains the data to be dragged, and specifies the format for the data. The drop target can examine the format to determine if it is compatible with the drop target.

     The component then calls the `mx.managers.DragManager.doDrag()` method, to initiate the drag-and-drop operation.
2. While the mouse button is still pressed, the user moves the mouse around the application. Flex displays the drag indicator image in your application.

   ***Note:** Releasing the mouse button when the drag indicator is not over a target ends the drag-and-drop operation. Flex generates a `DragComplete` event on the drag initiator, and the `DragManager.getFeedback()` method returns `DragManager.NONE`.*
3. If the user moves the drag indicator over a Flex component, Flex dispatches a `dragEnter` event for the component.
   - List-based components with `dropEnabled=true`

     Flex checks to see if the component can be a drop target.
   - Nonlist-based components, or list-based components with `dropEnabled=false`

     The component must define an event handler for the `dragEnter` event to be a drop target.

   The `dragEnter` event handler can examine the `DragSource` object to determine whether the data being dragged is in an accepted format. To accept the drop, the event handler calls the `DragManager.acceptDragDrop()` method. You must call the `DragManager.acceptDragDrop()` method for the drop target to receive the `dragOver`, `dragExit`, and `dragDrop` events.
   - If the drop target does not accept the drop, Flex examines all DisplayObject components under the mouse. Flex examines the components in order of depth to determine if any component accepts the drop data.
   - If the drop target accepts the drop, Flex dispatches the `dragOver` event as the user moves the drag indicator over the target.
4. (Optional) The drop target can handle the `dragOver` event. For example, the drop target can use this event handler to set the focus on itself. Or, the target can display a visual indication showing where the drag data is to be inserted.
5. (Optional) If the user decides not to drop the data onto the drop target and moves the drag indicator outside of the drop target without releasing the mouse button, Flex dispatches a `dragExit` event for the drop target. The drop target can optionally handle this event; for example, to undo any actions made in the `dragOver` event handler.
6. If the user releases the mouse while over the drop target, Flex dispatches a `dragDrop` event on the drop target.
   - List-based components with `dropEnabled=true`

     Flex automatically adds the drag data to the drop target.

- Nonlist-based components, or list-based components with `dropEnabled=false`

  The drop target must define an event listener for the `dragDrop` event handler to add the drag data to the drop target.

7 (Optional) When the drop operation completes, Flex dispatches a `dragComplete` event. The drag initiator can handle this event; for example, to delete the drag data from the drag initiator in the case of a move.

- List-based components with `dragEnabled=true`

  If this is a move operation, Flex automatically removes the drag data from the drag initiator.

- Nonlist-based components, or list-based components with `dragEnabled=false`

  The drag initiator completes any final processing required. If this was a move operation, the event handler must remove the drag data from the drag initiator. For an example of writing the event handler for the `dragComplete` event, see "Example: Moving and copying data for a nonlist-based control" on page 1929.

Example: Simple drag-and-drop operation for a nonlist-based control

The following example lets you set the background color of a Canvas container by dropping either of two colors onto it. You are not copying or moving any data; instead, you are using the two drag initiators as a color palette. You then drag the color from one palette onto the drop target to set its background color.

The drag initiators, two Canvas containers, implement an event handler for the `mouseDown` event to initiate the drag and drop operation. This is the only event required to be handled by the drag initiator. The drop target is required to implement event handlers for the `dragEnter` and `dragDrop` events.

```
<?xml version="1.0"?>
<!-- dragdrop\DandDCanvas.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    backgroundColor="white">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import mx.core.DragSource;
        import mx.managers.DragManager;
        import mx.events.*;
        import mx.containers.Canvas;
        // Initializes the drag and drop operation.
        private function mouseMoveHandler(event:MouseEvent):void {

            // Get the drag initiator component from the event object.
            var dragInitiator:Canvas=Canvas(event.currentTarget);

            // Get the color of the drag initiator component.
            var dragColor:int = dragInitiator.getStyle('backgroundColor');
            // Create a DragSource object.
            var ds:DragSource = new DragSource();
            // Add the data to the object.
            ds.addData(dragColor, 'color');
            // Call the DragManager doDrag() method to start the drag.
            DragManager.doDrag(dragInitiator, ds, event);
        }
        // Called when the user moves the drag indicator onto the drop target.
        private function dragEnterHandler(event:DragEvent):void {
            // Accept the drag only if the user is dragging data
            // identified by the 'color' format value.
            if (event.dragSource.hasFormat('color')) {
                // Get the drop target component from the event object.
                var dropTarget:Canvas=Canvas(event.currentTarget);
                // Accept the drop.
                DragManager.acceptDragDrop(dropTarget);
            }
        }

        // Called if the target accepts the dragged object and the user
        // releases the mouse button while over the Canvas container.
        private function dragDropHandler(event:DragEvent):void {
            // Get the data identified by the color format
            // from the drag source.
            var data:Object = event.dragSource.dataForFormat('color');
            // Set the canvas color.
            myCanvas.setStyle("backgroundColor", data);
        }
        ]]>
    </fx:Script>
    <!-- A horizontal box with red and green canvases that the user can drag. -->
    <mx:HBox>
        <mx:Canvas
            width="30" height="30"
            backgroundColor="red"
            borderStyle="solid"
            mouseMove="mouseMoveHandler(event);"/>
        <mx:Canvas
            width="30" height="30"
            backgroundColor="green"
            borderStyle="solid"
            mouseMove="mouseMoveHandler(event);"/>
    </mx:HBox>
    <mx:Label text="Drag a color onto the Canvas container."/>
    <!-- Handles dragEnter and dragDrop events to allow dropping. -->
    <mx:Canvas id="myCanvas"
        width="100" height="100"
        backgroundColor="#FFFFFF"
        borderStyle="solid"
        dragEnter="dragEnterHandler(event);"
        dragDrop="dragDropHandler(event);"/>

    <mx:Button id="b1"
        label="Clear Canvas"
        click="myCanvas.setStyle('backgroundColor', '0xFFFFFF');"/>
</s:Application>
```

The following sections describe the event handlers for the `mouseDown`, `dragEnter`, and `dragDrop` events.

Writing the mouseDown event handler

The event handler that initiates a drag-and-drop operation must do two things.

1 Create a DragSource object and initialize it with the drag data and the data format.

The DragSource object contains the drag data and a description of the drag data, called the data format. The event object for the `dragEnter` and `dragDrop` events contains a reference to this object in their `dragSource` property, which allows the event handlers to access the drag data.

You use the `DragSource.addData()` method to add the drag data and format to the DragSource object, where the `addData()` method has the following signature:

```
addData(data:Object, format:String):void
```

The `format` argument is a text string such as `"color"`, `"list data"`, or `"employee record"`. In the event handler for the `dragEnter` event, the drop target examines this string to determine whether the data format matches the type of data that the drop target accepts. If the format matches, the drop target lets users drop the data on the target; if the format does not match, the target does not enable the drop operation.

One example of using the format string is when you have multiple components in your application that function as drop targets. Each drop target examines the DragSource object during its `dragEnter` event to determine if the drop target supports that format. For more information, see "Handling the dragEnter event" on page 1908.

***Note:** List-based controls have predefined values for the `format` argument. For all list controls other than the Tree control, the format String is `"itemsByIndex"`. For the Tree control, the format String is `"treeItems"`. For previous versions of Flex, list-based controls used a format String of `"items"`. For more information, see "Using drag-and-drop with list-based controls" on page 1894.*

If you drag large or complex data items, consider creating a handler to copy the data, and specify it by calling the `DragSource.addHandler()` method instead of using the `DragSource.addData()` method. If you do this, the data does not get copied until the user drops it, which avoids the processing overhead of copying the data if a user starts dragging data but never drops it. The implementation of the list-based classes use this technique.

2 Call the `DragManager.doDrag()` method to start the drag-and-drop operation.

The `doDrag()` method has the following signature:

```
doDrag(dragInitiator:IUIComponent, dragSource:DragSource, mouseEvent:MouseEvent,
    dragImage:IFlexDisplayObject = null, xOffset:Number = 0, yOffset:Number = 0,
    imageAlpha:Number = 0.5, allowMove:Boolean = true):void
```

The `doDrag()` method requires three arguments: a reference to the component that initiates the drag operation (identified by the `event.currentTarget` object); the DragSource object that you created in step 1, and the event object passed to the event handler.

Optional arguments specify the drag indicator image and the characteristics of the image. For an example that specifies a drag indicator, see "Example: Specifying the drag indicator by using the DragManager" on page 1918.

Handling the dragEnter event

Flex generates a `dragEnter` event when the user moves the drag indicator over any component. A component *must* define a handler for a `dragEnter` event to be a drop target. The event handler typically performs the following actions:

- Use the `format` property in the DragSource object to determine whether the drag data is in a format accepted by the drop target.
- If the drag data is in a compatible format, the handler *must* call the `DragManager.acceptDragDrop()` method to enable the user to drop the data on the drop target.

- If the drag data is not in a compatible format, do not call the `DragManager.acceptDragDrop()` method. If the event handler does not call this method, the user cannot drop the data and the drop target will not receive the `dragOver`, `dragExit`, and `dragDrop` events.
- Optionally, perform any other actions necessary when the user first drags a drag indicator over a drop target.

The value of the `action` property of the event object for the `dragEnter` event is `DragManager.MOVE`, even if you are doing a copy. This is because the `dragEnter` event occurs before the drop target recognizes that the Control key is pressed to signal a copy.

The Flex default event handler for the `dragOver` event for a list-based control automatically sets the `action` property. For nonlist-based controls, or if you explicitly handle the `dragOver` event for a list-based control, use the `DragManager.showFeedback()` method to set the `action` property to a value that signifies the type of drag operation: `DragManager.COPY`, `DragManager.LINK`, `DragManager.MOVE`, or `DragManager.NONE`. For more information on the `dragOver` event, see "Example: Handling the dragOver and dragExit events for the drop target" on page 1924.

Handling the dragDrop event

The `dragDrop` event occurs when the user releases the mouse to drop data on a target, and the `dragEnter` event handler has called the `DragManager.acceptDragDrop()` method to accept the drop. You must define a handler for the event to add the drag data to the drop target.

The event handler uses the `DragSource.dataForFormat()` method to retrieve the drag data. In the previous example, the drag data contains the new background color of the drop target. The event handler then calls `setStyle()` to set the background color of the drop target.

Example: Handling drag and drop events in a list-based control

Flex automatically defines default event handlers for the drag-and-drop events when you set `dragEnabled` or `dropEnabled` property to `true` for an MX list-based control. You can either use these default event handlers, which requires you to do no additional work in your application, or define your own event handlers.

There are three common scenarios for using event handlers with the list-based controls:

Use the default event handlers When you set `dragEnabled` to `true` for a drag initiator, or when you set `dropEnabled` to `true` for a drop target, Flex handles all drag-and-drop events for you. You only have to define your own `dragDrop` event handler when you want to copy data as part of the drag-and-drop operation and the drop target uses a different data format. For more information, see "Moving and copying data" on page 1927.

Define your own event handlers If you want to control the drag-and-drop operation for a list-based control, you can explicitly handle the drag-and-drop events, just as you can for any component. In this scenario, set the `dragEnabled` property to `false` for a drag initiator, or set the `dropEnabled` property to `false` for a drop target. For more information on handling these events, see "Example: Simple drag-and-drop operation for a nonlist-based control" on page 1906.

Define your own event handlers and use the default event handlers You might want to add your own event handler for a drag-and-drop event, and also use the build in drag-and-drop handlers. In this case, your event handler executes first, then the default event handler provided by Flex executes. If, for any reason, you want to explicitly prohibit the execution of the default event handler, call the `Event.preventDefault()` method from within your event handler.

***Note:** If you call `Event.preventDefault()` in the event handler for the `dragComplete` or `dragDrop` event for a Tree control when dragging data from one Tree control to another, it prevents the drop.*

Because of the way data to a Tree control is structured, the Tree control handles drag and drop differently from the other list-based controls. For the Tree control, the event handler for the `dragDrop` event only performs an action when you move or copy data in the same Tree control, or copy data to another Tree control. If you drag data from one Tree control and drop it onto another Tree control to move the data, the event handler for the `dragComplete` event actually performs the work to add the data to the destination Tree control, rather than the event handler for the dragDrop event, and also removes the data from the source Tree control. This is necessary because to reparent the data being moved, Flex must remove it first from the source Tree control.

Therefore, if you call Event.preventDefault() in the event handler for the dragDrop or dragComplete events, you implement the drop behavior yourself. For more information, see "Example: Moving and copying data for a nonlist-based control" on page 1929.

The following example defines an event handler for the dragDrop event that accesses the data dragged from one DataGrid control to another. This event handler is executed before the default event handler for the dragDrop event to display in an Alert control the Artist field of each DataGrid row dragged from the drag initiator to the drop target:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleDGToDGAlert.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="650"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.DragEvent;
            import mx.controls.Alert;
            import mx.collections.ArrayCollection;

            private function initApp():void {
              srcgrid.dataProvider =  new ArrayCollection([
                {Artist:'Carole King', Album:'Tapestry', Price:11.99},
                {Artist:'Paul Simon', Album:'Graceland', Price:10.99},
                {Artist:'Original Cast', Album:'Camelot', Price:12.99},
                {Artist:'The Beatles', Album:'The White Album', Price:11.99}
              ]);

                destgrid.dataProvider = new ArrayCollection([]);
            }
            // Define the event listener.
            public function dragDropHandler(event:DragEvent):void {
                // dataForFormat() always returns an Vector.<Object>
                // for the list-based controls
                // in case multiple items were selected.
                var dragObj:Vector.<Object>=
                    event.dragSource.dataForFormat("itemsByIndex") as Vector.<Object>;
                // Get the Artist for all dragged albums.
                var artistList:String='';
                for (var i:Number = 0; i < dragObj.length; i++) {
                    artistList+='Artist: ' + dragObj[i].Artist + '\n';
                }

                Alert.show(artistList);
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:VGroup>
```

```
            <s:Label text="Available Albums"/>
            <mx:DataGrid id="srcgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true">
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"/>
                    <mx:DataGridColumn dataField="Album"/>
                    <mx:DataGridColumn dataField="Price"/>
                </mx:columns>
            </mx:DataGrid>
        </s:VGroup>
        <s:VGroup>
            <s:Label text="Buy These Albums"/>
            <mx:DataGrid id="destgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true"
                dragDrop="dragDropHandler(event);">
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"/>
                    <mx:DataGridColumn dataField="Album"/>
                    <mx:DataGridColumn dataField="Price"/>
                </mx:columns>
            </mx:DataGrid>
        </s:VGroup>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

Notice that the `dataForFormat()` method specifies an argument value of "`itemsByIndex`". This is because the list-based controls have predefined values for the data format of drag data. For all list controls other than the Tree control, the format String is "`itemsByIndex`". For the Tree control, the format String is "`treeItems`".

For Spark controls only, the DragSource object also contains the index of the item in the drag initiator that was clicked by the mouse. The format string of the index is “`caretIndex`”. The index is relative to the items in the drag data. Therefore, if the drag data contains three items, the index is a value between 0 and 2.

Notice that the return value of the `dataForFormat()` method is a Vector of type Object. Because list-based controls let you select multiple items, the `dataForFormat()` method always returns a Vector for a list-based control, even if you are only dragging a single item.

Using drag and drop with Flex applications running in AIR

When a Flex application runs in Adobe® AIR™, you can control whether the application uses the Flex drag manager or the AIR drag manager. These drag managers are implemented by the classes mx.managers.DragManager (Flex drag manager) and flash.desktop.NativeDragManager (AIR drag manager).

Internally, the Flex mx.managers.DragManager class uses an implementation class to determine which drag manager to use. It uses either the Flex mx.managers.DragManagerImpl class, or the AIR mx.managers.NativeDragManagerImpl class.

By default, an application defined by the Spark `<s:Application>` or the MX `<mx:Application>` tag uses the Flex drag-and-drop manager, even when the Flex application runs in AIR. If you run your Flex application in AIR, and you want to take advantage of the AIR drag-and-drop manager to drag and drop items from outside of AIR, then you must configure the Flex mx.managers.DragManager class to use the AIR drag-and-drop manager.

There are three scenarios that determine which drag-and-drop manager your Flex application uses when running in AIR:

1 Your main application file uses the Spark `<s:Application>` or the MX `<mx:Application>` tag. In this scenario, you use the Flex drag-and-drop manager, and cannot drag and drop items from outside of AIR.

2 Your main application file uses the Spark `<s:WindowedApplication>` or the MX `<mx:WindowedApplication>` tag. In this scenario, you use the AIR drag-and-drop manager, and can drag and drop items from outside of AIR.

3 Your main application file uses the Spark `<s:Application>` or the MX `<mx:Application>` tag, but loads the AIR drag-and-drop manager as represented by the mx.managers.NativeDragManagerImpl class. In this scenario, you use the AIR drag-and-drop manager, and can drag and drop items from outside of AIR.

For the third scenario, to use the AIR drag-and-drop manager in your Flex application, you must write your application to link to mx.managers.NativeDragManagerImpl class, and to load it at runtime, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initDandD();">

    <fx:Script>
        <![CDATA[
            // Ensure that the NativeDragManagerImpl class is linked in to your application.
            import mx.managers.NativeDragManagerImpl;
            var placeholder:NativeDragManagerImpl;

            // Handle the initialize event to load the DragManager.
            public function initDandD():void
            {
                // Ensure the DragManager is loaded, so that dragging in an AIR works.
                DragManager.isDragging;
            }
        ]]>
    </fx:Script>

    ...
</s:Application>
```

Two drag-and-drop events work differently depending on whether your application runs in Adobe® Flash® Player or AIR, as the following table shows:

Event	Flash Player	AIR
`dragEnter`	Triggers when you move the mouse pointer over any component during a drag and drop operation. The default value of the `DragEvent.action` property is `DragEvent.MOVE`.	Triggers when you move the mouse pointer over any component during a drag and drop operation. The default value of the `DragEvent.action` property is `DragEvent.COPY`.
`dragOver`	Triggers many times when you drag an item over a valid drop target.	Triggers many times when you drag an item over any component, even if the component is not a valid drop target.

There are several other differences between the Flex and AIR drag managers, as described below:

Characteristic	Flash player	AIR
Cursors	Flex draws the drag-and-drop cursors.	The operating system draws the cursors.
Styles	Cursor styles and the `mx.managers.DragManager.defaultDragImageSkin` property are supported.	Cursor styles and the `mx.managers.DragManager.defaultDragImageSkin` property are ignored.
Drag indicator	The drag image of the drag indicator can be an instance of the DisplayObject class, including an animated SWF file, an image, or a Flex component.	By default, the drag image is a bitmap created from the object being dragged. It does not change dynamically as the user drags it. If you pass a drag image to the `doDrag()` method, it must already be loaded by your application when you call `doDrag()`, otherwise it will be blank. For more information on the `doDrag()` method, see “Example: Specifying the drag indicator by using the DragManager” on page 1918.
Drop animation	The DragProxy class animates the drag image based on the results of the drop, such as accepted or rejected.	No custom animations can be used. The operating system handles the behavior of the cursor and the drag image.

Drag and drop differences between Spark and MX

Much of the drag and drop functionality is the same for the Spark and MX list-based controls. However, there are a few differences, as the following table shows:

Topic	Description	Spark	MX
Set the drag indicator	Specifies the appearance of the dragged data during the drag.	Defined by the optional `dragging` view state in the item renderer. You can further customize it by creating a subclass of the spark.components.supportClasses.ListItemDragProxy class.	Override the `mx.controls.listClasses.ListBase.dragImage` property. The default value is ListItemDragProxy.
Set the drop indicator	Specifies the appearance of the drop indicator which shows where the dragged data will be inserted into the drop target.	Define a `dropIndicator` skin part in the skin class of the drop target.	Use the `mx.controls.listClasses.ListBase.dropIndicatorSkin` property to set the skin class. The default value is ListDropIndicator.
Calculate the drop index	Returns the item index in the data provider of the drop target where the item will be dropped. Used by the `dragDrop` event handler to add the items in the correct location. Not available in the MX TileList or HorizontalList controls.	Use the `spark.layouts.supportClasses.LayoutBase.calculateDropLocation()` method. Use the `layout` property of the list class to access this method in the layout class of the component.	Use the `mx.controls.listClasses.ListBase.calculateDropIndex()` method.
Show drop feedback	Specifies to display the focus rectangle around the target control and positions the drop indicator where the drop operation should occur. If the control has active scroll bars, hovering the mouse pointer near the edges of the control scrolls the contents. You typically call this method from within the handler for the `dragOver` event. When you set `dropEnabled` to `true`, Flex automatically shows the drop indicator when necessary.	Use the `spark.layouts.supportClasses.LayoutBase.showDropIndicator()` method. Use the `layout` property to access this method in the layout class of the control. If you override the `dragEnter` event handler, and call `preventDefault()` so that the default handler does not execute, call `List.createDropIndicator()` to create the drop indicator.	Use the `mx.controls.listClasses.ListBase.showDropFeedback()` method.
Hide drop feedback	Hides drop target feedback and removes the focus rectangle from the target. You typically call this method from within the handler for the `dragExit` and `dragDrop` events. When you set `dropEnabled` to `true`, Flex automatically hides the drop indicator when necessary.	Use the `spark.layouts.supportClasses.LayoutBase.hideDropIndicator()` method. Use the `layout` property to access this method in the layout class of the control. Focus not handled by `hideDropIndicator().` If you override the `dragExit` event handler, and call `preventDefault()` so that the default handler does not execute, call `List.destroyDropIndicator()` to delete the drop indicator.	Use the `mx.controls.listClasses.ListBase.hideDropFeedback()` method.

Drag and drop examples

Example: Using a container as a drop target

To use a container as a drop target, you must use the `backgroundColor` property of the container to set a color. Otherwise, the background color of the container is transparent, and the Drag and Drop Manager is unable to detect that the mouse pointer is on a possible drop target.

In the following example, you use the Image control to load a draggable image into a Canvas container. You then add event handlers to let the user drag the Image control within the Canvas container to reposition it:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDImage.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            //Import classes so you don't have to use full names.
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;
            // Embed icon image.
            [Embed(source='assets/globe.jpg')]
            public var globeImage:Class;
            // The mouseMove event handler for the Image control
            // initiates the drag-and-drop operation.
            private function mouseMoveHandler(event:MouseEvent):void
            {
                var dragInitiator:Image=Image(event.currentTarget);
                var ds:DragSource = new DragSource();
                ds.addData(dragInitiator, "img");
                DragManager.doDrag(dragInitiator, ds, event);
            }

            // The dragEnter event handler for the Canvas container
            // enables dropping.
            private function dragEnterHandler(event:DragEvent):void {
                if (event.dragSource.hasFormat("img"))
                {
                    DragManager.acceptDragDrop(Canvas(event.currentTarget));
                }
            }
            // The dragDrop event handler for the Canvas container
            // sets the Image control's position by
            // "dropping" it in its new location.
            private function dragDropHandler(event:DragEvent):void {
                Image(event.dragInitiator).x =
                    Canvas(event.currentTarget).mouseX;
                Image(event.dragInitiator).y =
                    Canvas(event.currentTarget).mouseY;
            }
        ]]>
    </fx:Script>

    <!-- The Canvas is the drag target -->
    <mx:Canvas id="v1"
        width="500" height="500"
        borderStyle="solid"
        backgroundColor="#DDDDDD"
        dragEnter="dragEnterHandler(event);"
        dragDrop="dragDropHandler(event);">

        <!-- The image is the drag initiator. -->
        <s:Image id="myimg"
            source="@Embed(source='assets/globe.jpg')"
            mouseMove="mouseMoveHandler(event);"/>
    </mx:Canvas>
</s:Application>
```

Example: Setting the drag indicator for Spark controls

The drag indicator defines the appearance of the dragged data during drag-and-drop operations. The appearance of the dragged data is determined by the item renderer for the list-based control.

By default, Spark item renderers use the `normal` view state to display the dragged data. For more information on view states in Spark item renderers, see "Defining item renderer view states for a Spark container" on page 488.

Spark item renderers support the optional `dragging` view state that you can use to control the appearance of the drag indicator. The following item renderer add the `dragging` view state to display the dragged data in a bold, blue italic font:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- dragdrop\myComponents\MyListItemRenderer.mxml -->
<s:ItemRenderer focusEnabled="false"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:states>
        <s:State name="normal" />
        <s:State name="hovered" />
        <s:State name="selected" />
        <s:State name="normalAndShowsCaret"/>
        <s:State name="hoveredAndShowsCaret"/>
        <s:State name="selectedAndShowsCaret"/>
        <s:State name="dragging"/>
    </s:states>

    <s:Rect left="0" right="0" top="0" bottom="0">
        <s:stroke.normalAndShowsCaret>
            <s:SolidColorStroke
                color="{getStyle('selectionColor')}"
                weight="1"/>
        </s:stroke.normalAndShowsCaret>
        <s:stroke.hoveredAndShowsCaret>
            <s:SolidColorStroke
                color="{getStyle('selectionColor')}"
                weight="1"/>
        </s:stroke.hoveredAndShowsCaret>
        <s:stroke.selectedAndShowsCaret>
            <s:SolidColorStroke
                color="{getStyle('selectionColor')}"
                weight="1"/>
        </s:stroke.selectedAndShowsCaret>
        <s:fill>
            <s:SolidColor
                color.normal="0xFFFFFF"
                color.normalAndShowsCaret="0xFFFFFF"
                color.hovered="{getStyle('rollOverColor')}"
                color.hoveredAndShowsCaret="{getStyle('rollOverColor')}"
                color.selected="{getStyle('selectionColor')}"
                color.selectedAndShowsCaret="{getStyle('selectionColor')}"
                color.dragging="0xFFFFFF"/>
        </s:fill>
    </s:Rect>
    <s:Label id="labelDisplay" verticalCenter="0" left="3" right="3" top="6" bottom="4"
        fontStyle.dragging="italic" fontWeight.dragging="bold" color.dragging="blue"/>
</s:ItemRenderer>
```

The following application uses this item renderer:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToListMoveSparkDragIndicator.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            private function initApp():void {
                srclist.dataProvider =
                    new ArrayList(['Reading', 'Television', 'Movies']);
                destlist.dataProvider = new ArrayList([]);
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:VGroup>
            <s:Label text="Available Activities"/>
            <s:List id="srclist"
                allowMultipleSelection="true"
                dragEnabled="true"
                dragMoveEnabled="true"
                itemRenderer="myComponents.MyListItemRenderer"/>
        </s:VGroup>
        <s:VGroup>
            <s:Label text="Activities I Like"/>
            <s:List id="destlist"
                dropEnabled="true"/>
        </s:VGroup>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

Example: Specifying the drag indicator by using the DragManager

In the event handler for the `mouseDown` or `mouseUp` event, you can optionally specify a drag indicator in the `doDrag`() method of the DragManager class. If you do not specify a drag indicator, Flex uses a default drag indicator. The `doDrag()` method takes the following optional arguments to specify the drag indicator and its properties.

Argument	Description
dragIndicator	The image that defines the drag indicator. To specify a symbol, such as a JPEG image of a product that a user wants to order, use a string that specifies the symbol's name, such as myImage.jpg. To specify a component, such as a Flex container or control, create an instance of the control or container, configure and size it, and then pass it as an argument to the `doDrag()` method.
xOffset	Number that specifies the x offset, in pixels, for the `dragImage`. This argument is optional. If omitted, the drag indicator is shown at the upper-left corner of the drag initiator. The offset is expressed in pixels from the left edge of the drag indicator to the left edge of the drag initiator, and is usually a negative number.
yOffset	Number that specifies the y offset, in pixels, for the `dragImage`. This argument is optional. If omitted, the drag indicator is shown at the upper-left corner of the drag initiator. The offset is expressed in pixels from the top edge of the drag indicator to the top edge of the drag initiator, and is usually a negative number.
imageAlpha	A Number that specifies the alpha value used for the drag indicator image. If omitted, Flex uses an alpha value of 0.5. A value of 0 corresponds to transparent and a value of 1.0 corresponds to fully opaque.

You must specify a size for the drag indicator, otherwise it does not appear. The following example modifies the example in "Example: Using a container as a drop target" on page 1915 to use a 15 pixel by 15 pixel Image control as the drag indicator:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDImageProxy.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            //Import classes so you don't have to use full names.
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;
            // Embed icon image.
            [Embed(source='assets/globe.jpg')]
            public var globeImage:Class;
            // The mouseMove event handler for the Image control
            // initiates the drag-and-drop operation.
            private function mouseOverHandler(event:MouseEvent):void
            {
                var dragInitiator:Image=Image(event.currentTarget);
                var ds:DragSource = new DragSource();
                ds.addData(dragInitiator, "img");
                // The drag manager uses the Image control
                // as the drag indicator and sets the alpha to 1.0 (opaque),
                // so it appears to be dragged across the Canvas.
                var imageProxy:Image = new Image();
                imageProxy.source = globeImage;
                imageProxy.height=15;
                imageProxy.width=15;
                DragManager.doDrag(dragInitiator, ds, event,
                    imageProxy, -15, -15, 1.00);
            }
            // The dragEnter event handler for the Canvas container
            // enables dropping.
            private function dragEnterHandler(event:DragEvent):void {
                if (event.dragSource.hasFormat("img"))
                {
                    DragManager.acceptDragDrop(Canvas(event.currentTarget));
                }
            }
            // The dragDrop event handler for the Canvas container
            // sets the Image control's position by
            // "dropping" it in its new location.
            private function dragDropHandler(event:DragEvent):void {
                Image(event.dragInitiator).x =
                    Canvas(event.currentTarget).mouseX;
                Image(event.dragInitiator).y =
                    Canvas(event.currentTarget).mouseY;
            }
        ]]>
    </fx:Script>

    <!-- The Canvas is the drag target -->
    <mx:Canvas id="v1"
        width="500" height="500"
        borderStyle="solid"
        backgroundColor="#DDDDDD"
        dragEnter="dragEnterHandler(event);"
        dragDrop="dragDropHandler(event);">

        <!-- The image is the drag initiator. -->
        <s:Image id="myimg"
            source="@Embed(source='assets/globe.jpg')"
            mouseMove="mouseOverHandler(event);"/>
    </mx:Canvas>
</s:Application>
```

To use a control with specific contents, such as a VBox control with a picture and label, you must create a custom component that contains the control or controls, and use an instance of the component as the *dragIndicator* argument.

Example: Setting the drop indicator for Spark controls

The drop indicator shows where the dragged data will be inserted into the drop target. The appearance of the drop indicator is controlled by the skin class of the drop target. By default, the drop indicator for a Spark control is a solid line that spans the width of the control.

You can create a custom drop indicator by creating a custom skin class for the drop target. In your skin class, create a skin part named `dropIndicator` in the `<fx:Declarations>` area of the skin class, as the following example shows:

```
<fx:Declarations>
    <fx:Component id="dropIndicator">
        <s:Group includeInLayout="false"
            minWidth="4" minHeight="4"
            maxWidth="4" maxHeight="4">
            <s:Line xFrom="0" xTo="10" yFrom="5" yTo="5">
                <s:stroke>
                    <s:SolidColorStroke color="blue" weight="2"/>
                </s:stroke>
            </s:Line>
            <s:Line xFrom="5" xTo="10" yFrom="0" yTo="5">
                <s:stroke>
                    <s:SolidColorStroke color="blue" weight="2"/>
                </s:stroke>
            </s:Line>
            <s:Line xFrom="5" xTo="10" yFrom="10" yTo="5">
                <s:stroke>
                    <s:SolidColorStroke color="blue" weight="2"/>
                </s:stroke>
            </s:Line>
        </s:Group>
    </fx:Component>
</fx:Declarations>
```

This code shows an excerpt from the MyListSkin.mxml file, a custom skin class for the Spark List control. The lines define the drop indicator to be a blue arrow that appears in the drop target to indicate where the dragged data is added.

Note that the bounding Group container specifies the `maxWidth` and `maxHeight` properties. For the drop indicator, only the setting along the major axis of the control is applied. For example, for a List class using vertical layout, the x-axis is the major axis. Therefore, the `maxHeight` property is applied and the `maxWidth` property is ignored.

The Spark layout classes use the following rules to size and position the drop indicator:

1 The drop indicator's size is calculated to be as big as the gap between the neighboring data items in the control.

  Any minimum or maximum setting in the major axis of orientation is honored. Along the minor axis, minimum and maximum settings are ignored. The drop indicator is sized to be as wide as the respective column and as tall as the respective row.

2 After drop indicator is sized, it is centered within the gap between the data items.

The following example uses the MyListSkin.mxml skin class:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToListMoveSparkDropIndicator.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            import myComponents.MyListSkin;
            private function initApp():void {
                srclist.dataProvider =
                    new ArrayList(['Reading', 'Television', 'Movies']);
                destlist.dataProvider = new ArrayList([]);
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:VGroup>
            <s:Label text="Available Activities"/>
            <s:List id="srclist"
                allowMultipleSelection="true"
                dragEnabled="true"
                dragMoveEnabled="true"/>
        </s:VGroup>
        <s:VGroup>
            <s:Label text="Activities I Like"/>
            <s:List id="destlist"
                dropEnabled="true"
                skinClass="myComponents.MyListSkin"/>
        </s:VGroup>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

Example: Setting the cursor styles of the DragManager

The DragManager uses styles to control the display of the different cursors used during the drag and drop operation, such as the move and copy cursors. The cursors are defined as symbols in the Assets.swf file in the *flexInstallDir*\frameworks\projects\framework\assets directory.

By default, Flex defines the cursor styles as if you had used the following type selector in your application:

```
<fx:Style>
    @namespace mx "library://ns.adobe.com/flex/mx";

    mx|DragManager
    {
        copyCursor: Embed(source="Assets.swf",symbol="mx.skins.cursor.DragCopy");
        defaultDragImageSkin: ClassReference("mx.skins.halo.DefaultDragImage");
        linkCursor: Embed(source="Assets.swf",symbol="mx.skins.cursor.DragLink");
        moveCursor: Embed(source="Assets.swf",symbol="mx.skins.cursor.DragMove");
        rejectCursor: Embed(source="Assets.swf",symbol="mx.skins.cursor.DragReject");
    }
</fx:Style>
```

Use the `<fx:Style>` tag to define your own assets to use for the cursors. The following example replaces the copy cursor with a custom cursor:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToListMoveStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|DragManager
        {
            copyCursor: Embed(source="assets/globe.jpg");
        }
    </fx:Style>

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            private function initApp():void {
                srclist.dataProvider =
                    new ArrayCollection(['Reading', 'Television', 'Movies']);
                destlist.dataProvider = new ArrayCollection([]);
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:VGroup>
            <s:Label text="Available Activities"/>
            <s:List id="srclist"
                allowMultipleSelection="true"
                dragEnabled="true"
                dragMoveEnabled="true"/>
        </s:VGroup>
        <s:VGroup>
            <s:Label text="Activities I Like"/>
            <s:List id="destlist"
                dropEnabled="true"/>
        </s:VGroup>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

Example: Handling the dragOver and dragExit events for the drop target

The `dragOver` event occurs when the user moves the mouse over a drag-and-drop target whose `dragEnter` event handler has called the `DragManager.acceptDragDrop()` method. This event is dispatched continuously as the user drags the mouse over the target. The `dragOver` event handler is optional; you do not have to define it to perform a drag-and-drop operation.

The `dragOver` event is useful for specifying the visual feedback that the user gets when the mouse is over a drop target. For example, you can use the `DragManager.showFeedback()` method to specify the drag-feedback indicator that appears along with the drag indicator. This method uses four constant values for the argument, as the following table shows:

Argument value	**Icon**
`DragManager.COPY`	A green circle with a white plus sign indicating that you can perform the drop.
`DragManager.LINK`	A grey circle with a white arrow sign indicating that you can perform the drop.
`DragManager.MOVE`	A plain arrow indicating that you can perform the drop.
`DragManager.NONE`	A red circle with a white *x* appears indicating that a drop is prohibited. This is the same image that appears when the user drags over an object that is not a drag target.

You typically show the feedback indicator based on the keys pressed by the user during the drag-and-drop operation. The DragEvent object for the `dragOver` event contains Boolean properties that indicate whether the Control or Shift keys are pressed at the time of the event: `ctrlKey` and `shiftKey`, respectively. No key pressed indicates a move, the Control key indicates a copy, and the Shift key indicates a link. You then call the `showFeedback()` method as appropriate for the key pressed.

Another use of the `showFeedback()` method is that it determines the value of the `action` property of the DragEvent object for the `dragDrop`, `dragExit`, and `dragComplete` events. If you do not call the `showFeedback()` method in the `dragOver` event handler, the `action` property of the DragEvent is always set to `DragManager.MOVE`.

The `dragExit` event is dispatched when the user drags the drag indicator off the drop target, but does not drop the data onto the target. You can use this event to restore any visual changes that you made to the drop target in the `dragOver` event handler.

In the following example, you set the `dropEnabled` property of a List control to `true` to configure it as a drop target and to use the default event handlers. However, you want to provide your own visual feedback, so you also define event handlers for the `dragEnter`, `dragExit`, and `dragDrop` events. The `dragOver` event handler completely overrides the default event handler, so you call the `Event.preventDefault()` method to prohibit the default event handler from execution.

The `dragOver` event handler determines whether the user is pressing a key while dragging the drag indicator over the target, and sets the feedback appearance based on the key that is pressed. The `dragOver` event handler also sets the border color of the drop target to green to indicate that it is a viable drop target, and uses the `dragExit` event handler to restore the original border color.

For the `dragExit` and `dragDrop` handlers, you only want to remove any visual changes that you made in the `dragOver` event handlers, but otherwise you want to rely on the default Flex event handlers. Therefore, these event handlers do not call the `Event.preventDefault()` method:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDListToListShowFeedbackSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.events.DragEvent;
            import mx.managers.DragManager;
            import spark.layouts.supportClasses.DropLocation;

            private function initApp():void {
                firstList.dataProvider = new ArrayCollection([
                    {label:"First", data:"1"},
                    {label:"Second", data:"2"},
                    {label:"Third", data:"3"},
                    {label:"Fourth", data:"4"}
                ]);
                secondList.dataProvider = new ArrayCollection([]);
            }
            // Variable to store original border color.
            private var tempBorderColor:uint;

            // Flag to indicate that tempBorderColor has been set.
            private var borderColorSet:Boolean = false;
            private function dragOverHandler(event:DragEvent):void {

                // Explpicitly handle the dragOver event.
                event.preventDefault();

                // Since you are explicitly handling the dragOver event,
                // call showDropIndicator() to have the drop target
                // display the drop indicator.
                // The drop indicator is removed
                // automatically for the list controls by the built-in
                // event handler for the dragDrop event.
                var dropLocal:DropLocation =
                    event.currentTarget.layout.calculateDropLocation(event);
                event.currentTarget.layout.showDropIndicator(dropLocal);

                if (event.dragSource.hasFormat("itemsByIndex"))
                {
                    // Set the border to green to indicate that
                    // this is a drop target.
                    // Since the dragOver event is dispatched continuosly
                    // as you move over the drop target, only set it once.
                    if (borderColorSet == false) {
                        tempBorderColor =
                            event.currentTarget.getStyle('borderColor');
                        borderColorSet = true;
                    }

                    // Set the drag-feedback indicator based on the
                    // type of drag-and-drop operation.
                    event.currentTarget.setStyle('borderColor', 'green');
                    if (event.ctrlKey) {
                        DragManager.showFeedback(DragManager.COPY);
                        return;
                    }
                    else if (event.shiftKey) {
                        DragManager.showFeedback(DragManager.LINK);
                        return;
                    }
                    else {
                        DragManager.showFeedback(DragManager.MOVE);
                        return;
                    }
                }
                // Drag not allowed.
                DragManager.showFeedback(DragManager.NONE);
            }

            private function dragDropHandler(event:DragEvent):void {
                dragExitHandler(event);
            }
            // Restore the border color.
            private function dragExitHandler(event:DragEvent):void {
```

```
              event.currentTarget.setStyle('borderColor', tempBorderColor);
              borderColorSet = true;
            }
        ]]>
    </fx:Script>
    <s:HGroup id="myHG">
        <s:List  id="firstList"
            dragEnabled="true"
            dragMoveEnabled="true"/>
        <s:List  id="secondList"
            dropEnabled="true"
            dragOver="dragOverHandler(event);"
            dragDrop="dragExitHandler(event);"
            dragExit="dragExitHandler(event);"/>
    </s:HGroup>

    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

Moving and copying data

You implement a move and a copy as part of a drag-and-drop operation.

About moving data

When you move data, you add it to the drop target and delete it from the drag initiator. You use the `dragDrop` event for the drop target to add the data, and the `dragComplete` event for the drag initiator to remove the data.

How much work you have to do to implement the move depends on whether the drag initiator and drop target are list-based controls or nonlist-based controls:

List-based control You do not have to do any additional work; list-based controls handle all of the processing required to move data from one list-based control to another list-based control. For an example, see “Performing a drag and drop” on page 1894.

nonlist-based control If the drag initiator is a nonlist-based control, you have to implement the event handler for the `dragComplete` event to delete the drag data from the drag initiator. If the drop target is a nonlist-based control, you have to implement the event handler for the `dragDrop` event to add the data to the drop target. For an example, see “Example: Moving and copying data for a nonlist-based control” on page 1929.

About copying data

The list-based controls can automate all of the drag-and-drop operation except for when you copy the drag data to the drop target, and the drop target uses a different data format. If the drop target uses a different data format, you must explicitly handle the `dragDrop` event for the drop target.

When using a nonlist-based control as the drop target, you always have to write an event handler for the `dragDrop` event, regardless of whether you are performing a move or copy.

Copying data in an object-oriented environment is not a trivial task. An object may contain pointers to other objects that themselves contain pointers to even more objects. Rather than try to define a universal object copy as part of the drag-and-drop operation, Flex leaves it to you to implement object copying because you will have first-hand knowledge of your data format and requirements.

In some circumstances, you may have objects that implement a clone method that makes it easy to create a byte copy of the object. In other cases, you will have to perform the copy yourself by copying individual fields of the source object to the destination object.

Example: Copying data from an MX List control to an MX DataGrid control

You can use drag and drop to copy data between controls that use different data formats. To handle this situation, you write an event handler for the `dragDrop` event that converts the data from the format of the drag initiator to the format required by the drop target.

In the following example, you can move or copy data from a Spark List control to an MX DataGrid control. The event handler for the `dragDrop` event adds a new field to the dragged data that contains the date:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDListToDGSpark.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.DragEvent;
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.collections.IList;
            import mx.collections.ArrayCollection;

            private function initApp():void {
                srcList.dataProvider = new ArrayCollection([
                    {label:"First", data:"1"},
                    {label:"Second", data:"2"},
                    {label:"Third", data:"3"},
                    {label:"Fourth", data:"4"},
                ]);

                destDG.dataProvider = new ArrayCollection([]);
            }
            private function dragDropHandler(event:DragEvent):void {
                if (event.dragSource.hasFormat("itemsByIndex"))
                {
                    // Explicitly handle the dragDrop event.
                    event.preventDefault();

                    // Since you are explicitly handling the dragDrop event,
                    // call hideDropFeedback(event) to have the drop target
                    // hide the drop indicator.
                    // The drop indicator is created
                    // automatically for the list controls by the built-in
                    // event handler for the dragOver event.
                    event.currentTarget.hideDropFeedback(event);

                    // Get drop target.
                    var dropTarget:DataGrid =
                        DataGrid(event.currentTarget);
                    var itemsVector:Vector.<Object> =
                        event.dragSource.dataForFormat('itemsByIndex') as Vector.<Object>;
                    var tempItem:Object =
                        { label: itemsVector[0].label,
                            data: itemsVector[0].data,
                            date: new Date()
                        };

                    // Get the drop location in the destination.
                    var dropLoc:int = dropTarget.calculateDropIndex(event);

                    IList(dropTarget.dataProvider).addItemAt(tempItem, dropLoc);
                }
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:List  id="srcList"
            dragEnabled="true"
            dragMoveEnabled="true"/>
        <mx:DataGrid  id="destDG"
            dropEnabled="true"
            dragDrop="dragDropHandler(event);">
            <mx:columns>
                <mx:DataGridColumn dataField="label"/>
                <mx:DataGridColumn dataField="data"/>
                <mx:DataGridColumn dataField="date"/>
            </mx:columns>
        </mx:DataGrid>
    </s:HGroup>
    <s:Button id="b1"
        label="Reset"
        click="initApp();"/>
</s:Application>
```

Example: Moving and copying data for a nonlist-based control

The `dragComplete` event occurs on the drag initiator when a drag operation completes, either when the drag data drops onto a drop target, or when the drag-and-drop operation ends without performing a drop operation. The drag initiator can specify a handler to perform cleanup actions when the drag finishes, or when the target does not accept the drop.

One use of the `dragComplete` event handler is to remove from the drag initiator the objects that you move to the drop target. The items that you drag from a control are copies of the original items, not the items themselves. Therefore, when you drop items onto the drop target, you use the `dragComplete` event handler to delete them from the drag initiator.

To determine the type of drag operation (copy or move), you use the `action` property of the event object passed to the event handler. This method returns the drag feedback set by the `dragOver` event handler. For more information, see "Example: Handling the dragOver and dragExit events for the drop target" on page 1924.

In the following example, you drag an Image control from one Canvas container to another. As part of the drag-and-drop operation, you can move the Image control, or copy it by holding down the Control key. If you perform a move, the `dragComplete` event handler removes the Image control from its original parent container:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDImageCopyMove.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;
            // Embed icon image.
            [Embed(source='assets/globe.jpg')]
            public var globeImage:Class;
            // The mouseMove event handler for the Image control
            // functioning as the drag initiator.
            private function mouseOverHandler(event:MouseEvent):void
            {
                var dragInitiator:Image=Image(event.currentTarget);
                var ds:DragSource = new DragSource();
                ds.addData(dragInitiator, "img");
                // The drag manager uses the image as the drag indicator
                // and sets the alpha to 1.0 (opaque),
                // so it appears to be dragged across the canvas.
                var imageProxy:Image = new Image();
                imageProxy.source = globeImage;
                imageProxy.height=10;
                imageProxy.width=10;
                DragManager.doDrag(dragInitiator, ds, event,
                    imageProxy, -15, -15, 1.00);
            }

            // The dragEnter event handler for the Canvas container
            // functioning as the drop target.
            private function dragEnterHandler(event:DragEvent):void {
              if (event.dragSource.hasFormat("img"))
                DragManager.acceptDragDrop(Canvas(event.currentTarget));
            }

            // The dragOver event handler for the Canvas container
            // sets the type of drag-and-drop
            // operation as either copy or move.
            // This information is then used in the
            // dragComplete event handler for the source Canvas container.
            private function dragOverHandler(event:DragEvent):void
            {
                if (event.dragSource.hasFormat("img")) {
                    if (event.ctrlKey) {
                        DragManager.showFeedback(DragManager.COPY);
                        return;
                    }
                    else {
                        DragManager.showFeedback(DragManager.MOVE);
                        return;
                    }
                }
                DragManager.showFeedback(DragManager.NONE);
            }

            // The dragDrop event handler for the Canvas container
            // sets the Image control's position by
            // "dropping" it in its new location.
            private function dragDropHandler(event:DragEvent):void {
              if (event.dragSource.hasFormat("img")) {
                  var draggedImage:Image =
                    event.dragSource.dataForFormat('img') as Image;
                  var dropCanvas:Canvas = event.currentTarget as Canvas;

                  // Since this is a copy, create a new object to
                  // add to the drop target.
                  var newImage:Image=new Image();
                  newImage.source = draggedImage.source;
                  newImage.x = dropCanvas.mouseX;
                  newImage.y = dropCanvas.mouseY;
                  dropCanvas.addChild(newImage);
              }
            }

            // The dragComplete event handler for the source Canvas container
            // determines if this was a copy or move.
            // If a move, remove the dragged image from the Canvas.
            private function dragCompleteHandler(event:DragEvent):void {
                var draggedImage:Image =
                    event.dragInitiator as Image;
                var dragInitCanvas:Canvas =
                    event.dragInitiator.parent as Canvas;
                if (event.action == DragManager.MOVE)
                    dragInitCanvas.removeChild(draggedImage);
            }
        ]]>
    </fx:Script>
    <!-- Canvas holding the Image control that is the drag initiator. -->
    <mx:Canvas
```

```
        width="250" height="500"
        borderStyle="solid"
        backgroundColor="#DDDDDD">

    <!-- The Image control is the drag initiator and the drag indicator. -->
        <s:Image id="myimg"
            source="@Embed(source='assets/globe.jpg')"
            mouseMove="mouseOverHandler(event);"
            dragComplete="dragCompleteHandler(event);"/>
    </mx:Canvas>
    <!-- This Canvas is the drop target. -->
    <mx:Canvas
        width="250" height="500"
        borderStyle="solid"
        backgroundColor="#DDDDDD"
        dragEnter="dragEnterHandler(event);"
        dragOver="dragOverHandler(event);"
        dragDrop="dragDropHandler(event);">
    </mx:Canvas>
</s:Application>
```

Chapter 7: Enhancing usability

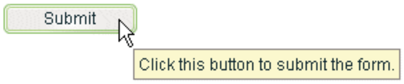
ToolTip controls

Adobe® Flex™ ToolTips are a flexible method of providing helpful information to your users. When a user moves the mouse pointer over graphical components, ToolTips pop up and text appears. You can use ToolTips to guide users through working with your application or customize them to provide additional functionality.

About ToolTip controls

ToolTips are a standard feature of many desktop applications. They make the application easier to use by displaying messages when the user moves the mouse pointer over an onscreen element, such as a Button control.

The following image shows ToolTip text that appears when the user hovers the mouse pointer over a button:



Click this button to submit the form.

When the user moves the mouse pointer away from the component or clicks the mouse button, the ToolTip disappears. If the mouse pointer remains over the component, the ToolTip eventually disappears. The default behavior is to display only one ToolTip at a time.

You can set the time it takes for the ToolTip to appear when a user moves the mouse pointer over the component. You can also set the amount of time it takes for the ToolTip to disappear.

If you define a ToolTip on a container, the ToolTipManager displays the parent's ToolTip for the children if the child does not have one.

Flex ToolTips support style sheets and the dynamic loading of ToolTip text. ToolTip text does not support embedded HTML. For more information on using style sheets and dynamic loading of text, see "Setting styles in ToolTips" on page 1936 and "Using dynamic ToolTip text" on page 1944.

Not all Spark text components support ToolTips. For example, you can add a ToolTip to the RichEditableText component, but not the RichText control.

Some components have their own ToolTip-like "data tips." These components include the List control, most chart controls, and DataGrid control. For more information, see that component's documentation.

ToolTips are the basis for the accessibility implementation within Flex. All controls that support accessibility do so by making their ToolTip text readable by screen readers such as JAWS. For more information on accessibility, see "Accessible applications" on page 2122.

Creating ToolTips

Every visual component that extends the UIComponent, UITextField, or UIFTETextField classes (which implement the IToolTipManagerClient interface) supports a `toolTip` property. This property is inherited from those class that implement IToolTipManagerClient. You set the value of the `toolTip` property to a text string, and when the mouse pointer hovers over that component, the text string appears. The following example sets the `toolTip` property text for a Button control:

```
<?xml version="1.0"?>
<!-- tooltips/BasicToolTip.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <s:Button id="b1" label="Click Me" toolTip="This Button does nothing."/>
</s:Application>
```

To set the value of a ToolTip in ActionScript, use the `toolTip` property of the component. The following example creates a new Button and sets the `toolTip` property of a Button control:

```
<?xml version="1.0"?>
<!-- tooltips/BasicToolTipActionScript.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
     public function createNewButton(event:MouseEvent):void {
        var myButton:Button = new Button();
        myButton.label = "Create Another Button";
        myButton.toolTip = "Click this new button to create another button.";
        myButton.addEventListener(MouseEvent.CLICK, createNewButton);
        addElement(myButton);
     }
    ]]>
  </fx:Script>

  <s:Button id="b1" label="Create Another Button"
        toolTip="Click this button to create another button."
        click="createNewButton(event);"/>
</s:Application>
```

If you do not define a ToolTip on the child of a container, the ToolTipManager displays the parent's ToolTip. For example, if you add a Button control to a Panel container that has a ToolTip, the user sees the Panel container's ToolTip text when the user moves the mouse pointer over the Panel. When the user moves the mouse pointer over the Button control, the Panel's ToolTip continues to be displayed. You can override the container's ToolTip text by setting the value of the child's `toolTip` property.

The following example shows the inheritance of ToolTip text:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipInheritance.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:VGroup toolTip="VGroup ToolTip">
        <s:Button id="b1" label="Button 1" toolTip="Button ToolTip"/>
        <s:Button id="b2" label="Button 2"/>
    </s:VGroup>

</s:Application>
```

When the mouse pointer is over button b1, the ToolTip displays “Button ToolTip”. When the mouse pointer is over button b2 or anywhere in the VGroup container except over button b1, the ToolTip displays “VGroup ToolTip”.

The TabNavigator container uses the ToolTips that are on its children. If you add a ToolTip to a child view of a TabNavigator container, the ToolTip appears when the mouse is over the tab for that view, but not when the mouse is over the view itself. If you add a ToolTip to the TabNavigator container, the ToolTip appears when the mouse is over either the tab or the view, unless the ToolTip is overridden by the tab or the view. ToolTips in the following controls also behave this way:

- Accordion
- ButtonBar
- LinkBar
- TabBar
- ToggleButtonBar

There is no limit to the size of the ToolTip text, although long messages can be difficult to read. When the ToolTip text reaches the width of the ToolTip box, the text wraps to the next line. You can add line breaks in ToolTip text. In ActionScript, you use the \n escaped newline character. In MXML tags, you use the &#13; XML entity.

The following examples show using the \n escaped newline character and the &#13; entity:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipNewlines.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="doSomething(event)">

    <fx:Script>
        <![CDATA[
            public function doSomething(event:Event):void {
                // Use the \n to force a line break in ActionScript.
                b1.toolTip = "Click this button \nto clear the form.";
            }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Clear" width="100"/>

    <!-- Use &#13; to force a line break in MXML tags. -->
    <s:Button id="b2" label="Submit" width="100"
        toolTip="Click this button &#13;to submit the form."/>
</s:Application>
```

You also have some flexibility in formatting the text of the ToolTip. You can apply styles and change other settings for all ToolTips in your application by using the ToolTip Cascading Style Sheets (CSS) type selector. The following sections describe how to set styles on the ToolTip text and box.

Setting styles in ToolTips

You can change the appearance of ToolTip text and the ToolTip box by using CSS syntax or the mx.styles.StyleManager class. Changes to ToolTip styles apply to all ToolTips in the current application.

The default styles for ToolTips are defined by the ToolTip type selector in the defaults.css file in the framework.swc file. You can use a type selector in the `<fx:Style>` tag to override the default styles of your ToolTips. The following example sets the styles of the ToolTip type selector using CSS syntax:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Style>
        @namespace mx "library://ns.adobe.com/flex/mx";
        mx|ToolTip {
            fontFamily: "Arial";
            fontSize: 19;
            fontStyle: "italic";
            color: #FF6699;
            backgroundColor: #33CC99;
        }
    </fx:Style>
    <s:Button id="b1" label="Click Me" toolTip="This Button does nothing."/>
</s:Application>
```

You can also use the StyleManager to apply styles to all ToolTips. You must first get a reference to the top-level StyleManager with the Application object’s `styleManager` property. You then get the ToolTip style declaration with the `getStyleDeclaration()` method. Finally, you apply a style to the ToolTip type selector with the `setStyle()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipStyleManager.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="setToolTipStyle()">
    <fx:Script>
        <![CDATA[
            private function setToolTipStyle():void {

styleManager.getStyleDeclaration("mx.controls.ToolTip").setStyle("fontStyle","italic");

styleManager.getStyleDeclaration("mx.controls.ToolTip").setStyle("fontSize","19");

styleManager.getStyleDeclaration("mx.controls.ToolTip").setStyle("fontFamily","Arial");

styleManager.getStyleDeclaration("mx.controls.ToolTip").setStyle("color","#FF6699");

styleManager.getStyleDeclaration("mx.controls.ToolTip").setStyle("backgroundColor","#33CC99"
);
            }
        ]]>
    </fx:Script>
  <s:Button id="b1" label="Click Me" toolTip="This Button does nothing."/>
</s:Application>
```

ToolTips use inheritable styles that you set globally. For example, you can set the `fontWeight` of ToolTips with the StyleManager by setting it on the global selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipGlobalStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="setToolTipStyle()">
    <fx:Script>
        <![CDATA[
            private function setToolTipStyle():void {
                styleManager.getStyleDeclaration("global").setStyle("fontWeight","bold");
                styleManager.getStyleDeclaration("global").setStyle("color","red");
            }
        ]]>
    </fx:Script>
  <s:Button id="b1" label="Click Me" toolTip="This Button does nothing, but the ToolTip is
styled red and bold."/>
</s:Application>
```

Styles set on the Application object typically apply to all UI objects. However, ToolTips do not inherit styles set on the Application object.

If you set the `fontWeight` property on the global selector, your change affects the text of many controls in addition to ToolTips, so be careful when using the global selector.

For a complete list of styles supported by ToolTips, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For more information on using styles, see “Styles and themes” on page 1492.

You can reskin ToolTip controls to give them an entirely new appearance. You can do this by using the ToolTipBorder programmatic skin or reskin them graphically. For an example of programmatically reskinning a ToolTip control, see "Reskinning ToolTips" on page 1958.

Setting the width of ToolTips

You can set the width of the ToolTip box by changing the `maxWidth` property of the ToolTip class. This property is static so when you set it, you are setting it for all ToolTip objects. You cannot set it on an instance of a ToolTip class.

The `maxWidth` property specifies the maximum width in pixels for new ToolTips boxes. For example, the following line changes the maximum width of the ToolTip box to 100 pixels:

```
<?xml version="1.0"?>
<!-- tooltips/SetMaxWidth.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
     import mx.controls.ToolTip;
     public function initApp():void {
        ToolTip.maxWidth = 100;
     }
     public function createNewButton(event:MouseEvent):void {
        var myButton:Button = new Button();
        myButton.label = "Create Another Button";
        myButton.toolTip = "Click this new button to create another button.";
        myButton.addEventListener(MouseEvent.CLICK, createNewButton);
        addElement(myButton);
     }
    ]]>
  </fx:Script>

  <s:Button id="b1" label="Create Another Button"
    click="createNewButton(event);"
    toolTip="Click this button to create a new one."/>
</s:Application>
```

Flex wraps the text of a ToolTip onto multiple lines to ensure that the width does not exceed this value. If the text in the ToolTip box is not as wide as the `maxWidth` property, Flex creates a box only wide enough for the text to fit.

The default value of `maxWidth` is `300`. If the value of `maxWidth` exceeds the width of the application, Flex clips the text in the ToolTip box.

Using ToolTip events

ToolTips trigger many events during their life cycle. These events are of type ToolTipEvent.

In addition to the `type` and `target` properties, the ToolTipEvent object references the ToolTip in its `toolTip` property. With a reference to the ToolTip, you can then access the ToolTip text with the `text` property.

To use events of type ToolTipEvent in your `<fx:Script>` blocks, you must import mx.events.ToolTipEvent class.

The following example plays a sound in response to the `TOOL_TIP_SHOW` event:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipsWithSoundEffects.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initApp()">
  <fx:Script>
    <![CDATA[
     import mx.events.ToolTipEvent;
     import flash.media.Sound;
     [Embed(source="../assets/sound1.mp3")]
     private var beepSound:Class;
     private var myClip:Sound;
     public function playSound():void {
        myClip.play();
     }
     private function myListener(event:ToolTipEvent):void {
        playSound();
     }
     private function initApp():void {
        myRichEditableText.addEventListener(ToolTipEvent.TOOL_TIP_SHOW, myListener);
        myClip = new beepSound();
     }
    ]]>
  </fx:Script>
  <s:RichEditableText id="myRichEditableText" toolTip="ToolTip" text="Mouse Over Me"/>
</s:Application>
```

Using ToolTips with MX navigator containers

MX NavBar and TabBar subclasses (such as ButtonBar, LinkBar, and ToggleButtonBar) support ToolTips in their data providers. The data provider array can contain a `toolTip` field that specifies the `toolTip` for the navigation items.

The following example creates ToolTips for each of the navigation items:

```
<?xml version="1.0"?>
<!-- tooltips/NavItemToolTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:ButtonBar>
        <s:dataProvider>
            <s:ArrayCollection>
                <fx:Object label="OK" toolTip="OK Button ToolTip"/>
                <fx:Object label="Cancel" toolTip="Cancel Button ToolTip"/>
            </s:ArrayCollection>
        </s:dataProvider>
    </s:ButtonBar>
    <mx:ButtonBar>
        <mx:dataProvider>
                <fx:Object label="OK" toolTip="OK Button ToolTip"/>
                <fx:Object label="Cancel" toolTip="Cancel Button ToolTip"/>
        </mx:dataProvider>
    </mx:ButtonBar>
</s:Application>
```

The Spark ButtonBar component does not by default support the use of a `toolTip` field in its data provider. To add ToolTips to a Spark ButtonBar in this way, you must create a custom item renderer. For more information, see "Custom Spark item renderers" on page 470.

You can use ToolTips on child elements in a navigator container's data provider. The component recognizes those ToolTips and displays them accordingly. In the following example, the ToolTips are propagated up to the LinkBar:

```
<?xml version="1.0"?>
<!-- tooltips/DataProviderToolTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:VBox>
     <!-- Create a LinkBar control to navigate the ViewStack. -->
     <mx:LinkBar dataProvider="{vs1}" borderStyle="solid"/>

     <!-- Define the ViewStack and the three child containers. -->
     <mx:ViewStack id="vs1" borderStyle="solid" width="100%" height="150">
        <mx:Canvas id="search" label="Search" toolTip="Search Screen">
           <mx:Label text="Search Screen"/>
        </mx:Canvas>
        <mx:Canvas id="custInfo" label="Customer"
           toolTip="Customer Info Screen">
           <mx:Label text="Customer Info"/>
        </mx:Canvas>
        <mx:Canvas id="accountInfo" label="Account"
           toolTip="Account Info Screen">
           <mx:Label text="Account Info"/>
        </mx:Canvas>
     </mx:ViewStack>
  </mx:VBox>
</s:Application>
```

You can also set the value of the NavBar's `toolTipField` property to point to the field in the data provider that provides a ToolTip. The data provider in the following example defines ToolTips in the `myToolTip` field:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipFieldExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:ButtonBar toolTipField="myToolTip">
    <mx:dataProvider>
     <fx:Object label="OK" myToolTip="OK Button TooTip"/>
     <fx:Object label="Cancel" myToolTip="Cancel Button ToolTip"/>
    </mx:dataProvider>
  </mx:ButtonBar>
</s:Application>
```

Using the ToolTip Manager

The ToolTipManager class lets you set basic ToolTip functionality, such as display delay and the disabling of ToolTips. It is located in the mx.managers package. You must import this class when using the ToolTipManager. The ToolTipManager class also contains a reference to the current ToolTip in its `currentToolTip` property.

Enabling and disabling ToolTips

You can enable and disable ToolTips in your applications. When you disable ToolTips, no ToolTip box appears when the user moves the mouse pointer over a visible component, regardless of whether that component's `toolTip` property is set.

You use the enabled property of the ToolTipManager to enable or disable ToolTips. You set this property to true to enable ToolTips or false to disable ToolTips. The default value is true.

The following example toggles ToolTips on and off when the user clicks the Toggle ToolTips button:

```
<?xml version="1.0"?>
<!-- tooltips/ToggleToolTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
     import mx.managers.ToolTipManager;
     private function toggleToolTips():void {
        if (ToolTipManager.enabled) {
           ToolTipManager.enabled = false;
        } else {
           ToolTipManager.enabled = true;
        }
     }
    ]]>
  </fx:Script>
  <s:Button id="b1"
    label="Toggle ToolTips"
    width="150"
    click="toggleToolTips();"
    toolTip="Click me to enable/disable tooltips."
  />

</s:Application>
```

Setting delay times

A *delay time* is a measurement of time that passes before something takes place. For example, after the user moves the mouse pointer over a component, there is a brief delay before the ToolTip appears. This gives someone who is not looking for ToolTip text enough time to move the mouse pointer away before seeing the pop-up.

The ToolTipManager lets you set the length of time that passes before the ToolTip box appears, and the length of time that a ToolTip remains on the screen when a mouse pointer hovers over the component. You can also set the delay between ToolTips.

You set the value of the ToolTipManager's showDelay, hideDelay, and scrubDelay properties in ActionScript. The following table describes the time delay properties of the ToolTipManager:

Property	Description
`showDelay`	The length of time, in milliseconds, that Flex waits before displaying the ToolTip box when a user moves the mouse pointer over a component that has a ToolTip. To make the ToolTip appear instantly, set the `showDelay` property to 0. The default value is 500 milliseconds, or half of a second.
`hideDelay`	The length of time, in milliseconds, that Flex waits to hide the ToolTip box after it appears. This amount of time only applies if the mouse pointer is over the target component. Otherwise the ToolTip disappears immediately when you move the mouse pointer away from the target component. After Flex hides a ToolTip box, the user must move the mouse pointer off the component and back onto it to see the ToolTip box again. If you set the `hideDelay` property to 0, Flex does not display the ToolTip. Adobe recommends that you use the default value of 10,000 milliseconds, or 10 seconds. If you set the `hideDelay` property to `Infinity`, Flex does not hide the ToolTip until the user triggers an event (such as moving the mouse pointer off the component). The following example sets the `hideDelay` property to `Infinity`: `ToolTipManager.hideDelay = Infinity;`
`scrubDelay`	The length of time, in milliseconds, that a user can take when moving the mouse between controls before Flex again waits for the duration of `showDelay` to display a ToolTip box. This setting is useful if the user moves quickly from one control to another; after displaying the first ToolTip, Flex displays the others immediately rather than waiting for the duration of the `showDelay` setting. The shorter the setting for `scrubDelay`, the more likely that the user must wait for an amount of time specified by the `showDelay` property in order to see the next ToolTip. A good use of this property is if you have several buttons on a toolbar, and the user will quickly scan across them to see brief descriptions of their functionality. The default value is 100.

The following example uses the Application control's `initialize` event to set the starting values for the ToolTipManager:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipTiming.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import mx.managers.ToolTipManager;
        private function initApp():void {
            ToolTipManager.enabled = true;// Optional. Default value is true.
            ToolTipManager.showDelay = 0;// Display immediately.
            ToolTipManager.hideDelay = 3000; // Hide after 3 seconds of being viewed.
        }
    ]]></fx:Script>
    <s:Button label="Click Me" toolTip="Click this Button to do something."/>
    <s:Button label="Click Me" toolTip="Click this Button to do something else."/>
    <s:Button label="Click Me" toolTip="Click this Button to do a third thing."/>
    <s:Button label="Click Me" toolTip="Click this Button to do the same thing."/>
</s:Application>
```

Using effects with ToolTips

You can use a custom effect or one of the standard Flex effects with ToolTips. You set the `showEffect` or `hideEffect` property of the ToolTipManager to define the effect that is triggered whenever a ToolTip is displayed or is hidden.

The following example uses the Fade effect so that ToolTips fade in when the user moves the mouse pointer over a component with a ToolTip:

```
<?xml version="1.0"?>
<!-- tooltips/FadeInToolTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600" height="600"
    initialize="app_init();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:Fade id="fadeIn" alphaFrom="0" alphaTo="1" duration="1000"/>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import mx.managers.ToolTipManager;
            public function app_init():void {
                ToolTipManager.showEffect = fadeIn;
            }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"
        toolTip="This is a ToolTip that fades in."/>
</s:Application>
```

By default, the font used in this example does not fade. You must use an embedded font to achieve the effect. For more information on using embedded fonts, see “Embed fonts” on page 1571.

If you set a fade out effect for the `hideEffect` event, the user must wait with the mouse hovering over the component to trigger that effect; the `hideToolTip` event is not triggered if the user moves the mouse pointer to a different component before the ToolTip object hides on its own.

For more information about using effects and defining custom effects, see “Introduction to effects” on page 1784.

Using dynamic ToolTip text

You can use ToolTips for more than just displaying static help text to the user. You can also bind the ToolTip text to data or component text. This lets you use ToolTips as a part of the user interface, showing drill-down information, query results, or more helpful text that is customized to the user experience.

You bind the value of the ToolTip text to the value of another control’s text using curly braces ({ }) syntax.

The following example inserts the value of the TextInput control into the Button control’s ToolTip text when the user moves the mouse pointer over the Button control:

```
<?xml version="1.0"?>
<!-- tooltips/BoundToolTipText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:TextInput id="txtTo" width="300"/>
    <s:Button label="Send" toolTip="Send e-mail to {txtTo.text}"/>
</s:Application>
```

In this example, if the user enters **fred@fred.com** in the TextInput box, and then moves the mouse pointer over the button, Flex displays the message “Send e-mail to fred@fred.com” in the ToolTip box.

Another approach to creating dynamic text for ToolTips is to intercept the ToolTip in its `toolTipShow` event handler and change the value of its `text` property. The following example registers the `myToolTipChanger()` method as a listener for the Button control’s `toolTipShow` event. The code in that method changes the value of the `ToolTipManager.currentToolTip.text` property to a value that is not known until run time.

```
<?xml version="1.0"?>
<!-- tooltips/DynamicToolTipText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initApp()">
  <fx:Script>
    <![CDATA[
     import mx.managers.ToolTipManager;
     import mx.controls.ToolTip;
     import mx.events.ToolTipEvent;
     import mx.core.FlexGlobals;
     public function initApp():void {
        b1.addEventListener(ToolTipEvent.TOOL_TIP_SHOW, myToolTipChanger)
     }

     public function myToolTipChanger(event:ToolTipEvent):void {
        /* Pass the value of myName in to your application as a flashVars variable.
           For example, on the query string. */
        ToolTipManager.currentToolTip.text = "Click the button, " +
           FlexGlobals.topLevelApplication.parameters.myName;
     }
    ]]>
  </fx:Script>
  <s:Button id="b1" label="Click Me" toolTip="Click the button."/>
</s:Application>
```

You can only create text for the current ToolTip in the `toolTipShow` event handler if the object had a ToolTip in the first place. For example, if the button in the previous example did not set a value for the `toolTip` property, no ToolTip would appear even after the `myToolTipChanger()` method was called.

For information about using the `FlexGlobals.topLevelApplication.application` object, see “Passing request data with flashVars properties” on page 229.

Creating custom ToolTips

The ToolTipManager has two methods that let you programmatically use ToolTips. These methods are `createToolTip()` and `destroyToolTip()`, which you use to create and destroy new ToolTip objects. When you create a ToolTip object, you can customize it as you would any object, with access to its properties, styles, events, and effects.

The `createToolTip()` method has the following signature:

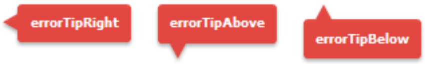
```
createToolTip(text:String, x:Number, y:Number, errorTipBorderStyle:String,
    context:IUIComponent):IToolTip
```

The *`text`* parameter defines the contents of the ToolTip.

The `x` and `y` parameters define the x and y coordinates of the ToolTip, relative to the application container.

The *`errorTipBorderStyle`* parameter sets the location of the pointer on the error tip. This parameter is optional. If you specify the value of the `errorTipBorderStyle` parameter in the `createToolTip()` method, Flex styles the ToolTip as an error tip. Valid values are `"errorTipRight"`, `"errorTipAbove"`, or `"errorTipBelow"`, and indicate the location of the error tip relative to the component. If you set the `errorTipBorderStyle` parameter to `null`, then the ToolTip is a normal ToolTip, not an error tip.

The following example shows the valid values and their resulting locations on the error tip:



The *`context`* parameter determines which StyleManager is used. Typically, you pass the object on which the ToolTip appears, so that the ToolTip's StyleManager is the same one use by that object. This object must be of type IUIComponent, so you might need to cast it in some cases, such as when you want to specify the context as the `event.currentTarget` in an event handler.

For more information about using error tips, see "Using error tips" on page 1954.

The `createToolTip()` method returns a new ToolTip object that implements the IToolTip interface. You typically cast the return value of this method to a ToolTip, although it is more efficient if you do not do this. To cast, you can do one of the following:

- Use the `as` keyword, as the following example shows:

  ```
  myTip = ToolTipManager.createToolTip(s,10,10) as ToolTip;
  ```

- Use the *type*`(`*object*`)` casting syntax, as the following example shows:

  ```
  myTip = ToolTip(ToolTipManager.createToolTip(s,10,10));
  ```

These methods of casting differ only in the way that they behave when a cast fails.

Flex displays the ToolTip until you destroy it. In general, you should not display more than one ToolTip box at a time, because it is confusing to the user.

You can use the `destroyToolTip()` method to destroy the specified ToolTip object. The `destroyToolTip()` method has the following signature:

```
destroyToolTip(toolTip:IToolTip):void
```

The *`toolTip`* parameter is the ToolTip object that you want to destroy. This is the object returned by the `createToolTip()` method.

The following example creates a custom ToolTip when you move the mouse over a Panel container that contains three Button controls. Each Button control has its own ToolTip that appears when you mouse over that particular control. The big ToolTip disappears only when you move the mouse away from the Panel container.

```
<?xml version="1.0"?>
<!-- tooltips/CreatingToolTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.managers.ToolTipManager;
            import mx.controls.ToolTip;
            import mx.core.IUIComponent;
            public var myTip:ToolTip;
            private function createBigTip(event:Event):void {
               var s:String = "These buttons let you save, exit, or continue with the current
operation."
                myTip =
ToolTipManager.createToolTip(s,75,75,null,IUIComponent(event.currentTarget)) as ToolTip;
                myTip.setStyle("backgroundColor",0xFFCC00);
                myTip.width = 125;
                myTip.height = 75;
            }
            private function destroyBigTip():void {
                ToolTipManager.destroyToolTip(myTip);
            }
        ]]>
    </fx:Script>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Panel {
            paddingLeft: 5;
            paddingRight: 5;
            paddingTop: 5;
            paddingBottom: 5;
        }
    </fx:Style>
    <s:Panel title="ToolTips" height="200" width="200" rollOver="createBigTip(event)"
rollOut="destroyBigTip()">
        <s:Button label="OK" y="20" toolTip="Save your changes and exit."/>
        <s:Button label="Apply" y="50" toolTip="Apply changes and continue."/>
        <s:Button label="Cancel" y="80" toolTip="Cancel and exit."/>
    </s:Panel>
</s:Application>
```

Customizing the current ToolTip

Another approach to customizing ToolTip objects is to intercept the current ToolTip and customize it rather than using the ToolTipManager to create a new ToolTip object.

You do this by customizing the ToolTip during one of the ToolTip-related events like `toolTipHide`, `toolTipShow`, and `toolTipShown`.

The following example triggers the customization on the ToolTip's `toolTipShown` event. This lets Flex handle all the logic for displaying and hiding the ToolTip, but lets you modify the current ToolTip's text, position, and other properties.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- tooltips/CustomToolTipInActionScript.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import spark.components.Button;
            import mx.core.IToolTip;
            import mx.events.ToolTipEvent;
            import mx.managers.ToolTipManager;
            import mx.core.FlexGlobals;

            public var myTip:IToolTip;
            public var b:Button;

            private function initApp():void {
                b = new Button();
                b.addEventListener("toolTipShown", createCustomTip);
                b.label = "Click Me";

                /* You must create a blank ToolTip so that the control
                   can dispatch ToolTip-related events. The new ToolTip
                   will replace this empty ToolTip. */
                b.toolTip = " ";
                addElement(b);
            }

            private function createCustomTip(e:ToolTipEvent):void {
                var s:String = "This is a ToolTip for the button.";
                myTip = ToolTipManager.currentToolTip;

                // Customize the text of the ToolTip.
                myTip.text = s;

                // Customize the alpha of the ToolTip.
                myTip.alpha = .6;

                // Customize the position of the ToolTip.
                myTip.x = FlexGlobals.topLevelApplication.mouseX + 20;
                myTip.y = FlexGlobals.topLevelApplication.mouseY;
            }
        ]]>
    </fx:Script>
</s:Application>
```

Implementing the IToolTip interface

You can also create a custom ToolTip by extending an existing control, such as a Panel or other container, and implementing the IToolTip interface. The following example uses a Panel container as the base for a new implementation of the IToolTip interface:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipComponents/PanelToolTip.mxml -->
<s:Panel
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    implements="mx.core.IToolTip"
    width="200"
    alpha=".75">
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var bodyText:String = "";
            //  Implement required methods of the IToolTip interface; these
            //  methods are not used in this example, though.
            public var _text:String;
            public function get text():String {
                return _text;
            }
            public function set text(value:String):void {
            }
        ]]>
    </fx:Script>
    <s:RichText text="{bodyText}" percentWidth="100"/>
</s:Panel>
```

In your application, you can create a custom ToolTip by intercepting the `toolTipCreate` event handler of the target component. In the event handler, you instantiate the new ToolTip and set its properties. You then point the `toolTip` property of the ToolTipEvent object to the new ToolTip.

In the following example, the first two buttons in the application use the custom PanelToolTip class in the ToolTipComponents package. The third button uses a default ToolTip to show you how the two are different. To run this example, store the PanelToolTip.mxml file in a subdirectory named ToolTipComponents.

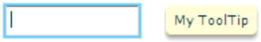
```
<?xml version="1.0"?>
<!-- tooltips/MainCustomApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import ToolTipComponents.PanelToolTip;
            import mx.events.ToolTipEvent;
         private function createCustomTip(title:String, body:String, event:ToolTipEvent):void
{
               var ptt:PanelToolTip = new PanelToolTip();
               ptt.title = title;
               ptt.bodyText = body;
               event.toolTip = ptt;
            }
        ]]>
    </fx:Script>

    <s:Button id="b1"
        label="Delete"
        toolTip=" "
        toolTipCreate="createCustomTip('DELETE','Click this button to delete the report.',
event)"
    />
    <s:Button id="b2"
        label="Generate"
        toolTip=" "
       toolTipCreate="createCustomTip('GENERATE','Click this button to generate the report.',
event)"
    />

    <s:Button id="b3"
        label="Stop"
        toolTip="Click this button to stop the creation of the report. This button uses a
standard ToolTip style."
    />
</s:Application>
```

Positioning custom ToolTips

When you use the ToolTipManager to create a custom ToolTip, you specify the coordinates of the ToolTip on the Stage. You do this by specifying the values of the `x` and `y` parameters of the new ToolTip in the `createToolTip()` method. These coordinates are relative to the Stage. For example, a value of 0,0 creates a ToolTip at the top-left corner of the application.

In some cases, you might not know the exact position that you want the ToolTip to be drawn in; instead, you want the location of the ToolTip to be relative to the target component (the component that has a ToolTip on it). In those cases, you can use the location of the target component to calculate the values of these coordinates. For example, if you want the ToolTip to appear to a component's right, you set the ToolTip's x position to be the x position of the component plus the component's width, plus some other value for an offset.

The following image shows the results of this formula:



You also set the value of the ToolTip's y position to be the same as the target component's y position to line the ToolTip and the component up horizontally.

One way to get the values you need to calculate the x position of the ToolTip is to use an event handler. Event objects passed to an event handler can give you the x position and the width of the target component.

The following example gets the value of the current target's `x`, `y`, and `width` properties in the focusIn event handler, and uses them to position the ToolTip. In this case, the current target is the TextInput control, and the ToolTip appears to its right with a 10-pixel offset.

```
<?xml version="1.0"?>
<!-- tooltips/PlacingToolTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="100" width="300">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
             import mx.controls.ToolTip;
             import mx.managers.ToolTipManager;
             import mx.core.IUIComponent;

             private var tip:ToolTip;
             private var s:String;

             private function showTip(event:Object):void {
                s="My ToolTip";
                // Position the ToolTip to the right of the current target.
                tip = ToolTipManager.createToolTip(s,
                    event.currentTarget.x + event.currentTarget.width + 10,
                    event.currentTarget.y, null,IUIComponent(event.currentTarget))
                    as ToolTip;
            }
            private function destroyTip(event:Object):void {
                 ToolTipManager.destroyToolTip(tip);
            }
        ]]>
    </fx:Script>
    <s:TextInput id="a"
        width="100"
        focusIn="showTip(event)"
        focusOut="destroyTip(event)"/>
    <s:TextInput id="b"
        width="100"
        focusIn="showTip(event)"
        focusOut="destroyTip(event)"/>
</s:Application>
```

The previous example creates a ToolTip on a target component that is not inside any containers. However, in many cases, your components will be inside layout containers such as a VGroup or an HGroup. Under these circumstances, the coordinates you access in the event handler will be relative to the container and not the main application. But the ToolTipManager expects global coordinates when positioning the ToolTip. This will position ToolTips in unexpected locations.

To avoid this, you can use the `contentToGlobal()` method to convert the coordinates in the event handler from local to global. All components that subclass UIComponent have this method. It takes a single Point that is relative to the target's enclosing container as an argument and returns a Point that is relative to the Stage.

The following example calls the TextInput control's `contentToGlobal()` method to convert the control's coordinates from those that are relative to the VGroup container to global coordinates.

```
<?xml version="1.0"?>
<!-- tooltips/PlacingToolTipsInContainers.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="400" width="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.controls.ToolTip;
            import mx.managers.ToolTipManager;
            import mx.core.IUIComponent;
            private var tip:ToolTip;
            private var s:String;

            private function showTipA(event:Object):void {
                s="My Tip A";
                tip = ToolTipManager.createToolTip(s,
                    event.currentTarget.x + event.currentTarget.width + 10,
                    event.currentTarget.y) as ToolTip;
            }
            private function showTipB(event:Object):void {
                s="My Tip B";
                var pt:Point = new Point(0,0);

                /* Call this method to convert the object's
                   coordinates inside its container to the stage's
                   global coordinates. */
                pt = event.currentTarget.contentToGlobal(pt);

                tip = ToolTipManager.createToolTip(s,
                    pt.x + event.currentTarget.width + 10, pt.y, null,
                    IUIComponent(event.currentTarget)) as ToolTip;
            }
            private function destroyTip(event:Object):void {
                 ToolTipManager.destroyToolTip(tip);
            }
        ]]>
    </fx:Script>
    <!-- A ToolTip at the top level. -->
    <!-- The event handler for this ToolTip does not use any special
    logic to account for whether the ToolTip is inside a container.
    But this ToolTip is not inside a container so it positions itself
    normally. -->
    <s:TextInput id="a"
        text="Good ToolTip placement"
        width="175"
        focusIn="showTipA(event)"
        focusOut="destroyTip(event)"/>
    <s:VGroup>
        <!-- A ToolTip inside a container. -->
        <!-- The event handler for this ToolTip accounts for the control
        being inside a container and positions the ToolTip using the
        contentToGlobal() method. -->
        <s:TextInput id="b"
            text="Good ToolTip placement"
            width="175"
            focusIn="showTipB(event)"
            focusOut="destroyTip(event)"/>
        <!-- A ToolTip inside a container. -->
        <!-- The event handler for this ToolTip does not use any special
        logic to account for whether the ToolTip is inside a container.
        As a result, it positions itself using coordinates that are relative
        to the container, but that are not converted to global coordinates. -->

        <s:TextInput id="c"
            text="Bad ToolTip placement"
            width="175"
            focusIn="showTipA(event)"
            focusOut="destroyTip(event)"/>
    </s:VGroup>
</s:Application>
```

Using error tips

Error tips are instances of the ToolTip class that get their styles from the `errorTip` CSS class selector. They are most often seen when the Flex validation mechanism displays a warning when data is invalid. But you can use the definitions of the `errorTip` style and apply it to ToolTips to create a custom validation warning mechanism.

You can trigger an error tip to appear on any control when you set its `errorString` property to anything other than an empty String (“”).

The styles of an error tip are defined in the defaults.css file, which is in the framework.swc file. It specifies the following default settings for `errorTip` (notice the period preceding `errorTip`, which indicates that it is a class selector):

```
.errorTip {
    color: #FFFFFF;
    fontSize: 9;
    fontWeight: "bold";
    shadowColor: #000000;
    borderColor: #CE2929;
    borderStyle: "errorTipRight";
    paddingBottom: 4;
    paddingLeft: 4;
    paddingRight: 4;
    paddingTop: 4;
}
```

You can customize the appearance of error tips by creating a new theme that overrides these styles, or by overriding the style properties in your application. For more information on creating themes, see “About themes” on page 1561.

You can also set the `errorColor` property of any UIComponent. This defines the color of the “halo” around a component that has an error tip. The ErrorSkin class handles the logic that draws this glow around the target component.

You can create ToolTips that look like validation error tips by applying the `errorTip` style to the ToolTip. The following example does not contain any validation logic, but shows you how to use the `errorTip` style to create ToolTips that look like validation error tips. When you run the example, press the Enter key after entering text into the TextInput controls.

```
<?xml version="1.0"?>
<!-- tooltips/ErrorTipStyle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
     import mx.controls.ToolTip;
     import mx.managers.ToolTipManager;
     import mx.core.IUIComponent;

     private var errorTip:ToolTip;
     private var myError:String;

     private function validateEntry(type:String, event:Object):void {
        if (errorTip) {
            resetApp();
        }

        // NOTE: Validation logic would go here.
        switch(type) {
           case "ssn":
            myError="Use SSN format (NNN-NN-NNNN)";
            break;
           case "phone":
            myError="Use phone format (NNN-NNN-NNNN)";
            break;
        }

        // Use the target's x and y positions to set position of error tip.
        trace("event.currentTarget.width" + event.currentTarget.width);
        trace("event.currentTarget.x" + event.currentTarget.x);

        errorTip = ToolTipManager.createToolTip(
          myError, event.currentTarget.x + event.currentTarget.width, event.currentTarget.y,
null, IUIComponent(event.currentTarget)) as ToolTip;

        // Apply the errorTip class selector.
        errorTip.setStyle("styleName", "errorTip");
     }

     private function resetApp():void {
        if (errorTip) {
            ToolTipManager.destroyToolTip(errorTip);
            errorTip = null;
        }
     }
    ]]>
  </fx:Script>
    <s:TextInput id="ssn" enter="validateEntry('ssn',event)"/>
    <s:TextInput id="phone" enter="validateEntry('phone',event)"/>
    <s:Label text="Press the enter key after entering text in each text input."/>
    <s:Button id="b1" label="Reset" click="resetApp()"/>
</s:Application>
```

Another way to use error tips is to set the value of the `errorString` property of the component. This causes the ToolTipManager to create an instance of a ToolTip and apply the `errorTip` style to that ToolTip without requiring any coding on your part.

The following example shows how to set the value of the `errorString` property to create an error tip without performing any validation:

```
<?xml version="1.0"?>
<!-- tooltips/ErrorString.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[
     import mx.controls.ToolTip;
     import mx.managers.ToolTipManager;

     private var errorTip:ToolTip;
     private var myError:String;

     private function validateEntry(type:String, event:Object):void {
        // NOTE: Validation logic would go here.
        switch(type) {
           case "ssn":
            myError="Use SSN format";
            break;
           case "phone":
            myError="Use phone format";
            break;
        }
        event.currentTarget.errorString = myError;
     }
    ]]>
  </fx:Script>
  <s:TextInput id="ssn" enter="validateEntry('ssn',event)"/>
  <s:TextInput id="phone" enter="validateEntry('phone',event)"/>
</s:Application>
```

You can also specify that the ToolTipManager create an error tip when you call the `createToolTip()` method. You do this by specifying the value of the `createToolTip()` method's `errorTipBorderStyle` argument (the fourth argument), as the following example shows:

```
<?xml version="1.0"?>
<!-- tooltips/CreatingErrorTips.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.managers.ToolTipManager;
            import mx.controls.ToolTip;
            import mx.core.IUIComponent;
            public var myTip:ToolTip;
            private function createBigTip(event:Event):void {
                var myError:String = "These buttons let you save, exit, or continue with the 
current operation.";
               // By setting the fourth argument ('errorTipBorderStyle:String') to a non-null 
value,
                // this ToolTip is created as an error tip.
                myTip = ToolTipManager.createToolTip(myError,
                    event.currentTarget.x + event.currentTarget.width,
                    event.currentTarget.y, "errorTipRight",
                    IUIComponent(event.currentTarget)) as ToolTip;
            }
            private function destroyBigTip():void {
                ToolTipManager.destroyToolTip(myTip);
            }
        ]]>
    </fx:Script>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|Panel {
            paddingLeft: 5;
            paddingRight: 5;
            paddingTop: 5;
            paddingBottom: 5;
        }
    </fx:Style>
    <s:Panel id="myPanel" title="ToolTips" rollOver="createBigTip(event)" 
rollOut="destroyBigTip()">
        <s:Button label="OK" y="20" toolTip="Save your changes and exit."/>
        <s:Button label="Apply" y="50" toolTip="Apply changes and continue."/>
        <s:Button label="Cancel" y="80" toolTip="Cancel and exit."/>
    </s:Panel>
</s:Application>
```

Note that if you set the value of the `errorTipBorderStyle` parameter to null, then a regular ToolTip, and not an error tip, is created.

For more information on using validators, see “Validating Data” on page 1964.

Reskinning ToolTips

You can apply a programmatic skin to ToolTip controls. This method of applying skins is consistent with the Halo component architecture. It does not use the rules that apply to the Spark component architecture.

ToolTip skins are defined by the ToolTipBorder programmatic skin. This file is located in the mx.skins.halo package.

To reskin ToolTips, you edit the ToolTipBorder class file, and then apply the new skin to the ToolTip by using CSS. For more information on skinning, see "Skinning MX components" on page 1655.

Reskin ToolTips by using CSS

1 Open the mx.skins.halo.ToolTipBorder.as file. This file is included with the source files, as described in "MX component skin resources" on page 1662.

2 Save the ToolTipBorder.as file under another name, and in a different location. The filename must be the same as the new class name.

3 Change the package from mx.skins.halo to your package name, or to the empty package, as the following example shows:

```
//package mx.skins.halo { // Old package name
package { // New, empty package
```

4 Change the class name in the file, as the following example shows:

```
//public class ToolTipBorder extends RectangularBorder // Old name
public class MyToolTipBorder extends RectangularBorder // New name
```

5 Change the constructor to reflect the new class name, as the following example shows:

```
//public function ToolTipBorder() // Old constructor
public function MyToolTipBorder() // New constructor
```

6 Comment out the versioning line, as the following example shows:

```
//include "../../core/Version.as";
```

7 Edit the class file to change the appearance of the ToolTip. You do this by editing the "toolTip" case in the `updateDisplayList()` method. That is the method that draws the ToolTip's border and sets the default styles. Most commonly, you change the arguments of the `drawRoundRect()` method to change the appearance of the ToolTip.

The following example adds a red tinge to the background of the ToolTip's box by replacing the default `backgroundColor` property with an array of colors:

```
var highlightAlphas:Array = [0.3,0.0];
drawRoundRect(3, 1, w-6, h-4, cornerRadius, [0xFF0000, 0xFFFFBB],
    backgroundAlpha);
```

The values of the `cornerRadius` and `backgroundAlpha` properties that are shown in the previous code snippet are set earlier in the `updateDisplayList()` method.

8 Save the class file.

9 In your application, edit the ToolTip's `borderSkin` style property to point to the new skin. You can do this in an `<fx:Style>` tag inside your application, or by using external CSS file or a custom theme file. The following example sets the `borderSkin` property to the new skin class:

```
<?xml version="1.0"?>
<!-- skins/ApplyCustomToolTipSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Style>
     @namespace mx "library://ns.adobe.com/flex/mx";

     mx|ToolTip {
        borderSkin: ClassReference("MyToolTipBorder");
     }
  </fx:Style>
  <s:Button id="b1" label="Click Me" toolTip="Click this button"/>
</s:Application>
```

You must specify the full package name in the CSS reference. In this example, the file MyToolTipBorder.as is in an empty package and, therefore, has no package designation such as mx.skins.halo.

10 Compile and run the application with the new skin file in the source path. You do this by setting the value of the compiler's `source-path` option.

If the skin class is in the same directory as the application, you do not have to add its location to the source path. The current directory is assumed. For more information, see "Flex compilers" on page 2164.

Cursor Manager

The Cursor Manager in Adobe® Flex® lets you control the cursor image in your Flex application. You can use the Cursor Manager to provide visual feedback to users to indicate when to wait for processing to complete, to indicate allowable actions, or to provide other types of feedback. The cursor image can be a JPEG, GIF, PNG, or SVG image, a Sprite object, or a SWF file.

About the Cursor Manager

By default, Flex uses the system cursor as the application cursor. You control the system cursor by using the settings of your operating system.

The Flex Cursor Manager lets you control the cursor image in your Flex application. For example, if your application performs processing that requires the user to wait until the processing completes, you can change the cursor so that it reflects the waiting period. In this case, you can change the cursor to an hourglass or other image.

You also can change the cursor to provide feedback to the user to indicate the actions that the user can perform. For example, you can use one cursor image to indicate that user input is enabled, and another to indicate that input is disabled.

You can use a JPEG, GIF, PNG, or SVG image, a Sprite object, or a SWF file as the cursor image.

Creating and removing a cursor

To use the Cursor Manager, you import the mx.managers.CursorManager class into your application, and then reference its static properties and methods.

The Cursor Manager controls a prioritized list of cursors, where the cursor with the highest priority is currently visible. If the cursor list contains more than one cursor with the same priority, the Cursor Manager displays the most recently created cursor.

You create a new cursor, and set an optional priority for the cursor, by using the static `setCursor()` method of the CursorManager class. This method adds the new cursor to the cursor list. If the new cursor has the highest priority, it is displayed immediately. If the priority is lower than a cursor already in the list, it is not displayed until the cursor with the higher priority is removed.

To remove a cursor from the list, you use the static `removeCursor()` method. If the cursor is the currently displayed cursor, the Cursor Manager displays the next cursor in the list, if one exists. If the list ever becomes empty, the Cursor Manager displays the default system cursor.

The `setCursor()` method has the following signature:

```
public static setCursor(cursorClass:Class, priority:int = 2, xOffset:Number = 0,
yOffset:Number = 0):int
```

The following table describes the arguments for the `setCursor()` method:

Argument	**Description**	**Req/Opt**
cursorClass	The class name of the cursor to display.	Required
priority	The priority level of the cursor. Valid values are `CursorManagerPriority.HIGH`, `CursorManagerPriority.MEDIUM`, and `CursorManagerPriority.LOW`. The default value is 2, corresponding to `CursorManagerPriority.MEDIUM`.	Optional
xOffset	The x offset of the cursor relative to the mouse pointer. The default value is 0.	Optional
yOffset	The y offset of the cursor relative to the mouse pointer. The default value is 0.	Optional

This method returns the ID of the new cursor. You pass the ID to the `removeCursor()` method to delete the cursor. This method has the following signature:

```
static removeCursor(cursorID:int):void
```

***Note:** In AIR, each instance of the Window class uses its own instance of the CursorManager class. In an AIR application, instead of directly referencing the static methods and properties of the CursorManager class, use the `Window.cursorManager` property to reference the CursorManager instance for the Window instance.*

The following example changes the cursor to a custom *wait* or *busy* cursor while a large image file loads. After the load completes, the application removes the busy cursor and returns the cursor to the system cursor.

***Note:** The Flex framework includes a default busy cursor. For information on using this cursor, see "Using a busy cursor" on page 1962.*

```
<?xml version="1.0"?>
<!-- cursors\CursorManagerApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.managers.CursorManager;
            import flash.events.*;

            // Define a variable to hold the cursor ID.
            private var cursorID:Number = 0;

            // Embed the cursor symbol.
            [Embed(source="assets/wait.jpg")]
            private var waitCursorSymbol:Class;

            // Define event listener to display the wait cursor
            // and to load the image.
            private function initImage(event:MouseEvent):void {
                // Set busy cursor.
                cursorID = CursorManager.setCursor(waitCursorSymbol);
                // Load large image.
                image1.source = "assets/DSC00034.JPG";
            }
            // Define an event listener to remove the wait cursor.
            private function loadComplete(event:Event):void {
                CursorManager.removeCursor(cursorID);
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <!-- Image control to load the image. -->
        <s:Image id="image1"
            height="50"
            width="100"
            complete="loadComplete(event);"/>
        <!-- Button triggers the load. -->
        <s:Button id="myButton" label="Show" click="initImage(event);"/>
    </s:VGroup>
</s:Application>
```

This example uses a JPEG image as the cursor image. You can also use a Sprite, or a PNG, GIF, SVG, or SWF file, as the following example shows:

```
[Embed(source="assets/wait.swf")]
var waitCursorSymbol:Class;
```

Or you can reference a symbol from a SWF file, as the following example shows:

```
[Embed(source="assets/cursorList.swf", symbol="wait")]
var waitCursorSymbol:Class;
```

An advantage of using a SWF file is that you can create an animated cursor.

Using a busy cursor

Flex defines a default busy cursor that you can use to indicate that your application is processing, and that users should wait until that processing completes before the application will respond to inputs. The default busy cursor is an animated clock.

You can control a busy cursor in several ways:

- You can use Cursor Manager methods to set and remove the busy cursor.
- You can use the `showBusyCursor` property of the SWFLoader, WebService, HttpService, and RemoteObject classes to automatically display the busy cursor.

Setting a busy cursor

The following static Cursor Manager methods control the busy cursor:

Method	Description
`setBusyCursor()`	Displays the busy cursor.
`removeBusyCursor()`	Removes the busy cursor from the cursor list. If other busy cursor requests are still active in the cursor list, which means that you called the `setBusyCursor()` method more than once, a busy cursor does not disappear until you remove all busy cursors from the list.

You can modify the example in "Creating and removing a cursor" on page 1959 to use the default busy cursor, as the following example shows:

```
<?xml version="1.0"?>
<!-- cursors\DefBusyCursorApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.managers.CursorManager;
            import flash.events.*;

            private function initImage(event:MouseEvent):void {
                CursorManager.setBusyCursor();
                image1.source = "assets/DSC00034.JPG";
            }

            private function loadComplete(event:Event):void {
                CursorManager.removeBusyCursor();
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <!-- Image control to load the image. -->
        <s:Image id="image1"
            height="50"
            width="100"
            complete="loadComplete(event);"/>

        <!-- Button triggers the load. -->
        <s:Button id="myButton" label="Show" click="initImage(event);"/>
    </s:VGroup>
</s:Application>
```

Setting the busy cursor does not prevent a user from interacting with your application; a user can still enter text or select buttons. However, all containers support the `enabled` property. By default, this property is set to `true` to enable user interaction with the container and with the container's children. If you set the `enabled` property to `false` when you display a busy cursor, Flex dims the color of the container and of all of its children, and blocks user input to the container and to all of its children.

You can also disable user interaction for the entire application by setting the `FlexGlobals.topLevelApplication.enabled` property to `false`. If you are in a subclass or an ActionScript-only application, you must either explicitly import the mx.core.FlexGlobals class, or specify `mx.core.FlexGlobals.topLevelApplication.enabled` to set the property's value.

The busy cursor has a priority of `CursorManagerPriority.LOW`. Therefore, if the cursor list contains a cursor with a higher priority, the busy cursor does not appear until you remove the higher-priority cursor. To create a default busy cursor at a higher priority level, use the `setCursor()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- cursors\ShowBusyCursorAppHighP.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.managers.CursorManager;
            import mx.managers.CursorManagerPriority;
            import flash.events.*;

            // Define a variable to hold the cursor ID.
            private var cursorID:Number = 0;

            // Define event listener to display the busy cursor
            // and to load the image.
            private function initImage(event:MouseEvent):void {
                // Set busy cursor.
                cursorID = CursorManager.setCursor(
styleManager.getStyleDeclaration("mx.managers.CursorManager").getStyle("busyCursor"),CursorM
anagerPriority.HIGH);
                // Load large image.
                image1.source = "assets/DSC00034.JPG";
            }
            // Define an event listener to remove the wait cursor.
            private function loadComplete(event:Event):void {
                CursorManager.removeCursor(cursorID);
            }
        ]]>
    </fx:Script>
    <s:VGroup>
        <!-- Image control to load the image. -->
        <s:Image id="image1"
            height="50"
            width="100"
            complete="loadComplete(event);"/>
        <!-- Button triggers the load. -->
        <s:Button id="myButton" label="Show" click="initImage(event);"/>
    </s:VGroup>
</s:Application>
```

This statement uses the `getStyleDeclaration()` method of the top-level StyleManager (accessed with the Application object's `styleManager` property) to get the CSSStyleDeclaration object for the Cursor Manager, and uses this object's `getStyle()` method to get the busy cursor, which it sets as a high priority cursor.

When you use this technique, you must also use the cursor ID in the `removeCursor()` method to remove the busy cursor.

Using the showBusyCursor property

The SWFLoader and MX Image controls, and the WebService, HTTPService, and RemoteObject classes, have a `showBusyCursor` property that automatically displays the default busy cursor until the class completes loading data. The default value is `false`.

The Spark Image control does not have a `showBusyCursor` property.

If you set the `showBusyCursor` property to `true`, Flex displays the busy cursor when the first `progress` event of the control is triggered, and hides the busy cursor when the `complete` event is triggered. The following example shows how you can simplify the example in the section "Setting a busy cursor" on page 1962 by using the `showBusyCursor` property of the MX Image control:

```
<?xml version="1.0"?>
<!-- cursors\ShowBusyCursorApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:VGroup>
        <!-- MX Image control to load the image. Note that the Spark
            Image control does not support the showBusyCursor property. -->
        <mx:Image id="image1"
            height="50"
            width="100"
            scaleContent="true"
            showBusyCursor="true"/>

        <!-- Button triggers the load. -->
        <s:Button id="myButton" label="Show"
            click="image1.load('../assets/DSC00034.JPG');"/>
    </s:VGroup>
</s:Application>
```

Validating Data

You use the Adobe® Flex™ data validation mechanism to validate the data in an application. Flex provides predefined validators for many common types of user-supplied data, such as date, number, and currency values.

Validating data

The data that a user enters in a user interface might or might not be appropriate to the application. In Flex, you use a *validator* to ensure the values in the fields of an object meet certain criteria. For example, you can use a validator to ensure that a user enters a valid phone number value, to ensure that a String value is longer than a set minimum length, or ensure that a ZIP code field contains the correct number of digits.

In typical client-server environments, data validation occurs on the server after data is submitted to it from the client. One advantage of using Flex validators is that they execute on the client, which lets you validate input data before transmitting it to the server. By using Flex validators, you eliminate the need to transmit data to and receive error messages back from the server, which improves the overall responsiveness of your application.

***Note:** Flex validators do not eliminate the need to perform data validation on the server, but provide a mechanism for improving performance by performing some data validation on the client.*

Flex includes a set of validators for common types of user input data, including the following:

- Validating credit card numbers
- Validating currency
- Validating dates
- Validating e-mail addresses
- Validating numbers
- Validating phone numbers
- Validating using regular expressions
- Validating social security numbers
- Validating strings
- Validating ZIP codes

You often use Flex validators with data models. For more information about data models, see “Storing data” on page 889.

About validators

You define validators by using MXML or ActionScript. In MXML, you declare a validator in an `<fx:Declarations>` tag. You define validators in an `<fx:Declarations>` tag because they are not visual components.

For example, to declare the standard PhoneNumberValidator validator, you use the `<mx:PhoneNumberValidator>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidator.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define the PhoneNumberValidator. -->
        <mx:PhoneNumberValidator id="pnV"
            source="{phoneInput}" property="text"/>
    </fx:Declarations>
    <!-- Define the TextInput control for entering the phone number. -->
    <s:TextInput id="phoneInput"/>
    <s:TextInput id="zipCodeInput"/>
</s:Application>
```

In the previous example, you enter a value into the TextInput control for the phone number. When you remove focus from the TextInput control by selecting the TextInput control for the ZIP code, the validator executes.

You use the source property of the validator to specify an object, and the property property to specify a field of the object to validate. For more information on the `source` and `property` properties, see “About the source and property properties” on page 1967.

In the previous example, the validator ensures that the user enters a valid phone number in the TextInput control. A valid phone number contains at least 10 digits, plus additional formatting characters. For more information, see “Validating phone numbers” on page 2000.

You declare validators in ActionScript either in a script block within an MXML file, or in an ActionScript file, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidatorAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Import PhoneNumberValidator.
            import mx.validators.PhoneNumberValidator;
            // Create the validator.
            private var v:PhoneNumberValidator = new PhoneNumberValidator();
            private function createValidator():void {
                // Configure the validator.
                v.source = phoneInput;
                v.property = "text";
            }
        ]]>
    </fx:Script>
    <!-- Define the TextInput control for entering the phone number. -->
    <s:TextInput id="phoneInput" creationComplete="createValidator();"/>
    <s:TextInput id="zipCodeInput"/>
</s:Application>
```

Spark and MX validators

Flex defines two sets of validators: Spark and MX. The Spark validators rely on the classes in the flash.globalization package. The flash.globalization classes use the locale data provided by the operating system. Therefore, Spark validators provide behavior that is consistent with the operating system and has access to all the locales supported by the operating system.

MX validators use the Flex ResourceManager to access locale-specific data from properties files that are included in the Flex SDK. The MX validators provide the same behavior across operating systems, but are limited to the locales provided by the Flex SDK or by the application developer.

The following table lists the Spark and MX validators. When possible, Adobe recommends that you use the Spark validators in your application:

Spark validator	**MX validator**	**Description**
	CreditCardValidator	Validate a credit card number.
CurrencyValidator	CurrencyValidator	Validate a currency value.
	DateValidator	Validate a date and time value.
	EmailValidator	Validate an email address.
NumberValidator	NumberValidator	Validate a numeric value.
	PhoneNumberValidator	Validate a phone number.
	SocialSecurityValidator	Validate a social security number.
	StringValidator	Validate a String value.
	ZipCodeValidator	Validate a U.S. or Canadian postal code.

Benefits of Spark validators

The Spark validators provides the following functionality:

- Locale-specific parsing of numbers and currency amounts.
- Locale-specific string comparison. For more information, see "Sorting and matching" on page 2062.
- Locale-specific uppercase and lowercase string conversion.
- Support non-European digits in a validation
- Validation of negative and positive formats of all numbers and currencies.
- Consistency with other applications running on the system.
- Operating system updates that include new locales or changes to existing locales are automatically supported.

The Spark validators use of the `locale` style property to select a locale. The `locale` style is an inheritable style that you can set for the entire application or specifically for a particular validator. Once set, the `locale` governs the validation provided by these classes.

If a Spark validator does not explicitly set the `locale` style, then it uses the value specified by the `locale` style of the application container. If you do not set the `locale` style property, the application uses the global default from the defaults.css style sheet, which defaults to `en`. For more information, see "Setting the locale" on page 2056.

You can explicitly configure the validator to use the default locale by setting the `locale` style to the constant value `flash.globalization.LocaleID.DEFAULT`.

More Help topics

"Setting the locale" on page 2056

"Formatting dates, numbers, and currencies" on page 2057

About the source and property properties

Validators use the following two properties to specify the item to validate:

source Specifies the object containing the property to validate. Set this to an instance of a component or a data model. You use data binding syntax in MXML to specify the value for the `source` property. This property supports dot-delimited Strings for specifying nested properties.

property A String that specifies the name of the property of `source` that contains the value to validate.

You can set these properties in any of the following ways:

- In MXML when you use a validator tag.
- In ActionScript by assigning values to the properties.
- When you call the `validate()` method to invoke a validator programmatically. For more information, see "Triggering validation programmatically" on page 1973.

You often specify a Flex user-interface control as the value of the `source` property, and a property of the control to validate as the value of the `property` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ZCValidator.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            source="{myZip}"
            property="text"/>
    </fx:Declarations>

    <s:TextInput id="phoneInput"/>
    <s:TextInput id="myZip"/>
</s:Application>
```

In this example, you use the Flex ZipCodeValidator to validate the data entered in a TextInput control. The TextInput control stores the input data in its `text` property.

About triggering validation

You trigger validation either automatically in response to an event, or programmatically by an explicit call to the `validate()` method of a validator.

When you use events, you can cause the validator to execute automatically in response to a user action. For example, you can use the `click` event of a Button control to trigger validation on the fields of a form, or the `valueCommit` event of a TextInput control to trigger validation after a user enters information in the control. For more information, see "Triggering validation by using events" on page 1969.

You can also trigger a validation programmatically. For example, you might have to inspect multiple, related input fields to perform a single validation. Or you might have to perform conditional validation based on a user input. For example, you may allow a user to select the currency used for payment, such as U.S. dollars or Euros. Therefore, you want to make sure that you invoke a validator configured for the specified currency. In this case, you can make an explicit call to the `validate()` method to trigger the correct validator for the input value. For more information, see "Triggering validation programmatically" on page 1973.

About validating required fields

Flex validators can determine when a user enters an incorrect value into a user-interface control. In addition, all validators support the `required` property, which, if `true`, specifies that a missing or empty value in a user-interface control causes a validation error. The default value is `true`. Therefore, a validation error occurs by default if the user fails to enter any required data in a control associated with a validator. To disable this check, set the `required` property to `false`. For more information, see "Validating required fields" on page 1981.

About validation errors

If a validation error occurs, by default Flex draws a red box around the component associated with the failure. If the user moves the pointer over the component, Flex displays the error message associated with the error.

***Note:** For the Spark Form container, the error message appears in the help area of the Form container without you having to move the pointer over the component.*

You can customize the look of the component and the error message associated with the error. For more information on validation errors, see “Working with validation errors” on page 1986.

About validation events

Validation is event driven. You can use events to trigger validation, programmatically create and configure validators in response to events, and listen for events dispatched by validators.

For example, when a validation operation completes, a validator dispatches a `valid` or `invalid` event, depending on the results of the validation. You can listen for these events, and then perform any additional processing that your application requires.

Alternatively, Flex components dispatch `valid` and `invalid` events, depending on the results of the validation. This lets you listen for the events being dispatched from the component being validated, rather than listening for events dispatched by the validator.

You are not required to listen for validation events. By default, Flex handles a failed validation by drawing a red box around the control that is associated with the source of the data binding. For a successful validation, Flex clears any indicator of a previous failure. For more information, see “Working with validation events” on page 1990.

Using validators

Triggering validation by using events

You can trigger validators automatically by associating them with an event. In the following example, the user enters a ZIP code in a TextInput control, and then triggers validation by clicking the Button control:

```
<?xml version="1.0"?>
<!-- validators\ZCValidatorTriggerEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            source="{myZip}"
            property="text"
            trigger="{mySubmit}"
            triggerEvent="click"/>
    </fx:Declarations>
    <s:TextInput id="myZip"/>
    <s:Button id="mySubmit" label="Submit"/>
</s:Application>
```

This example uses the `trigger` and `triggerEvent` properties of the ZipCodeValidator class to associate an event with the validator. These properties have the following values:

trigger Specifies the component generating the event that triggers the validator. If omitted, by default Flex uses the value of the `source` property.

triggerEvent Specifies the event that triggers the validation. If omitted, Flex uses the `valueCommit` event. Flex dispatches the `valueCommit` event whenever the value of a control changes. Usually this is when the user removes focus from the component, or when a property value is changed programmatically. If you want a validator to ignore all events, set `triggerEvent` to an empty string (`""`).

For information on specific validator classes, see "Using standard validators" on page 1993.

Triggering validation by using the default event

You can rewrite the example from the previous section to use default values for the trigger and triggerEvent properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ZCValidatorDefEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            source="{myZip}"
            property="text"/>
    </fx:Declarations>
    <s:TextInput id="myZip"/>
    <s:Button id="mySubmit" label="Submit"/>
</s:Application>
```

By omitting the `trigger` and `triggerEvent` properties, Flex triggers the validator when the TextInput control dispatches the `valueCommit` event. Flex controls dispatch the `valueCommit` event when its values changes by user interaction or programmatically.

Triggering validation for data bindings

Data binding provides a syntax for automatically copying the value of a property of one object to a property of another object at run time. With data binding, Flex copies the source value to the destination, typically in response to a modification to the source. The source and destination of data bindings are typically Flex components or data models.

In the next example, you bind data entered in a TextInput control to a data model so that the data in the TextInput control is automatically copied to the data model:

```
<?xml version="1.0"?>
<!-- validators\ValWithDataBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define a data model for storing the phone number. -->
        <fx:Model id="userInfo">
            <phoneInfo>
                <phoneNum>{phoneInput.text}</phoneNum>
            </phoneInfo>
        </fx:Model>
    </fx:Declarations>
    <!-- Define the TextInput control for entering the phone number. -->
    <s:TextInput id="phoneInput"/>
    <s:TextInput id="zipCodeInput"/>
</s:Application>
```

You can use a validator along with a data binding to validate either the source or destination of the data binding, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerWithDataBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define a data model for storing the phone number. -->
        <fx:Model id="userInfo">
            <phoneInfo>
                <phoneNum>{phoneInput.text}</phoneNum>
            </phoneInfo>
        </fx:Model>
        <!-- Define the PhoneNumberValidator. -->
        <mx:PhoneNumberValidator id="pnV"
            source="{phoneInput}"
            property="text"/>
    </fx:Declarations>
    <!-- Define the TextInput control for entering the phone number. -->
    <s:TextInput id="phoneInput"/>
    <s:TextInput id="zipCodeInput"/>
</s:Application>
```

This example uses a PhoneNumberValidator to validate the data entered in the TextInput control. In this example, the following occurs:

- You assign the validator to the source of the data binding.
- You use the default event, `valueCommit`, on the TextInput control to trigger the validator. This means the validator executes when the user removes focus from the TextInput control by selecting the TextInput control for the ZIP code.

- Flex updates the destination of the data binding on every change to the source. This means that the `userInfo.phoneNum` field updates on every change to the TextInput control, while the validator executes only when the user removes focus from the TextInput control to trigger the `valueCommit` event. You can use the validator's `triggerEvent` property to specify a different event to trigger the validation.

In a model-view-controller (MVC) design pattern, you isolate the model from the view and controller portions of the application. In the previous example, the data model represents the model, and the TextInput control and validator represents the view.

The TextInput control is not aware that its data is bound to the data model, or that there is any binding on it at all. Since the validator is also assigned to the TextInput control, you have kept the model and view portions of your application separate and can modify one without affecting the other.

However, there is nothing in Flex to prohibit you from assigning a validator to the data model, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerWithDataBindingOnModel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define a data model for storing the phone number. -->
        <fx:Model id="userInfo">
            <phoneInfo>
                <phoneNum>{phoneInput.text}</phoneNum>
            </phoneInfo>
        </fx:Model>

        <!-- Define the PhoneNumberValidator. -->
        <mx:PhoneNumberValidator id="pnV"
            source="{userInfo}"
            property="phoneNum"
            trigger="{phoneInput}"
            listener="{phoneInput}"/>
    </fx:Declarations>
    <!-- Define the TextInput control for entering the phone number. -->
    <s:TextInput id="phoneInput"/>
    <s:TextInput id="zipCodeInput"/>
</s:Application>
```

In this example, you trigger the data validator by using the `valueCommit` event of the TextInput control, but assign the validator to a field of the data model, rather than to a property of the TextInput control.

This example also uses the `listener` property of the validator. This property configures the validator to display validation error information on the specified object, rather than on the source of the validation. In this example, the source of the validation is a model, so you display the visual information on the TextInput control that provided the data to the model. For more information, see "Specifying a listener for validation" on page 1989.

If the model has a nesting structure of elements, you use dot-delimited Strings with the `source` property to specify the model element to validate, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerWithDataBindingComplexModel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define a data model for storing the phone number. -->
        <fx:Model id="userInfo">
            <user>
                <phoneInfo>
                    <homePhoneNum>{homePhoneInput.text}</homePhoneNum>
                    <cellPhoneNum>{cellPhoneInput.text}</cellPhoneNum>
                </phoneInfo>
            </user>
        </fx:Model>

        <!-- Define the PhoneNumberValidator. -->
        <mx:PhoneNumberValidator id="hPNV"
            source="{userInfo.phoneInfo}"
            property="homePhoneNum"
            trigger="{homePhoneInput}"
            listener="{homePhoneInput}"/>

        <!-- Define the PhoneNumberValidator. -->
        <mx:PhoneNumberValidator id="cPNV"
            source="{userInfo.phoneInfo}"
            property="cellPhoneNum"
            trigger="{cellPhoneInput}"
            listener="{cellPhoneInput}"/>
    </fx:Declarations>
    <!-- Define the TextInput controls for entering the phone number. -->
    <s:Label text="Home Phone:"/>
    <s:TextInput id="homePhoneInput"/>
    <s:Label text="Cell Phone:"/>
    <s:TextInput id="cellPhoneInput"/>
</s:Application>
```

Triggering validation programmatically

All validators define a `validate()` method that you can call to invoke a validator directly, rather than triggering the validator automatically by using an event.

The `validate()` method has the following signature:

```
validate(value:Object = null, supressEvents:Boolean = false):ValidationResultEvent
```

The arguments have the following values:

value If *`value`* is `null`, use the `source` and `property` properties to specify the data to validate. If `value` is non-null, it specifies a field of an object relative to the `this` keyword, which means an object in the scope of the document.

You should also set the `Validator.listener` property when you specify the `value` argument. When a validation occurs, Flex applies visual changes to the object specified by the listener property. By default, Flex sets the `listener` property to the value of the `source` property. However, because you do not specify the `source` property when you pass the value argument, you should set it explicitly. For more information, see “Specifying a listener for validation” on page 1989.

supressEvents If false, dispatch either the valid or invalid event on completion. If true, do not dispatch events.

This method returns an event object containing the results of the validation that is an instance of the ValidationResultEvent class. For more information on using the return result, see “Handling the return value of the validate() method” on page 1974.

In the following example, you create and invoke a validator programmatically in when a user clicks a Button control:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerProg.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Import ZipCodeValidator.
            import mx.validators.ZipCodeValidator;
            private var v:ZipCodeValidator = new ZipCodeValidator();
            private function performValidation():void {
                v.domain = "US or Canada";
                // Set the listener property to the component
                // used to display validation errors.
                v.listener=myZip;
                v.validate(myZip.text);
            }
        ]]>
    </fx:Script>
    <s:TextInput id="myZip"/>
    <s:Button label="Submit" click="performValidation();"/>
</s:Application>
```

Notice that you are still using an event to trigger the performValidation() function that creates and invokes the validator, but the event itself does not automatically invoke the validator.

Any errors in the validator are shown on the associated component, just as if you had triggered the validation directly by using an event.

Handling the return value of the validate() method

You may want to inspect the return value from the validate() method to perform some action when the validation succeeds or fails. The validate() method returns an event object with a type defined by the ValidationResultEvent class.

The ValidationResultEvent class defines several properties, including the type property. The type property of the event object contains either ValidationResultEvent.VALID or ValidationResultEvent.INVALID, based on the validation results. You can use this property as part of your validation logic, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerProgProcessReturnResult.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Import the ValidationResultEvent class.
            import mx.events.ValidationResultEvent;
            import mx.validators.ZipCodeValidator;
            public var v:ZipCodeValidator = new ZipCodeValidator();
            // Define variable for storing the validation event object.
            public var vResult:ValidationResultEvent;
            public function performValidation():void {
                v.domain = "US or Canada";
                v.listener=myZip;
                vResult = v.validate(myZip.text);

                if (vResult.type==ValidationResultEvent.VALID) {
                    // Validation succeeded.
                    myTA.text='OK';
                }
                else {
                    // Validation failed.
                    myTA.text='Fail';
                }
            }
        ]]>
    </fx:Script>

    <s:TextInput id="myZip"/>
    <s:Button label="Submit" click="performValidation();"/>

    <s:TextArea id="myTA"/>
</s:Application>
```

The ValidationResultEvent class has additional properties that you can use when processing validation events. For more information, see “Working with validation events” on page 1990.

Triggering the DateValidator and CreditCardValidator

The DateValidator and CreditCardValidator can validate multiple fields by using a single validator. A CreditCardValidator examines one field that contains the credit card number and a second field that contains the credit card type. The DateValidator can examine a single field that contains a date, or multiple fields that together make up a date.

When you validate an object that contains multiple properties that are set independently, you often cannot use events to automatically trigger the validator because no single field contains all of the information required to perform the validation.

One way to validate a complex object is to call the `validate()` method of the validator based on some type of user interaction. For example, you might want to validate the multiple fields that make up a date in response to the `click` event of a Button control, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\DateAndCC.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define the data model. -->
        <fx:Model id="date">
            <dateInfo>
                <month>{monthInput.text}</month>
                <day>{dayInput.text}</day>
                <year>{yearInput.text}</year>
            </dateInfo>
        </fx:Model>

        <!-- Define the validators. -->
        <mx:DateValidator id="dayV"
            triggerEvent=""
            daySource="{dayInput}"
            dayProperty="text"
            monthSource="{monthInput}"
            monthProperty="text"
            yearSource="{yearInput}"
            yearProperty="text"/>
    </fx:Declarations>

    <!-- Define the form to populate the model. -->
    <s:Form>
        <s:TextInput id="monthInput"/>
        <s:TextInput id="dayInput"/>
        <s:TextInput id="yearInput"/>
    </s:Form>
    <!-- Define the button to trigger validation. -->
    <s:Button label="Submit"
        click="dayV.validate();"/>
</s:Application>
```

The validator in this example examines all three input fields. If any field is invalid, validation fails. The validator highlights only the invalid fields that failed. For more information on how validators signal validation errors, see “Working with validation errors” on page 1986. For more information on the DateValidator and CreditCardValidator, see “Using standard validators” on page 1993.

Invoking multiple validators in a function

You can invoke multiple validators programmatically from a single function. In this example, you use the ZipCodeValidator and PhoneNumberValidator validators to validate the ZIP code and phone number input to a data model.

```
<?xml version="1.0"?>
<!-- validators\ValidatorCustomFuncStaticVals.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            // Import event class.
            import mx.events.ValidationResultEvent;

            // Define variable for storing the validation event object.
            private var vResult:ValidationResultEvent;

            private function validateZipPhone():void {
                // Validate the ZIP code.
                vResult = zipV.validate();
                // If the ZIP code is invalid,
                // do not move on to the next field.
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

                // Validate the phone number.
                vResult = pnV.validate();
                // If the phone number is invalid,
                // do not move on to the validation.
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:Model id="person">
            <userInfo>
                <zipCode>{zipCodeInput.text}</zipCode>
                <phoneNumber>{phoneNumberInput.text}</phoneNumber>
            </userInfo>
        </fx:Model>
        <!-- Define the validators. -->
        <mx:ZipCodeValidator id="zipV"
            source="{zipCodeInput}"
            property="text"/>
        <mx:PhoneNumberValidator id="pnV"
            source ="{phoneNumberInput}"
            property="text"/>
    </fx:Declarations>

    <s:Form>
        <s:FormItem>
            <s:TextInput id="zipCodeInput"/>
        </s:FormItem>
        <s:FormItem>
            <s:TextInput id="phoneNumberInput"/>
        </s:FormItem>
    </s:Form>

    <s:Button label="Validate"
        click="validateZipPhone();"/>
</s:Application>
```

In this example, you use the predefined ZipCodeValidator and PhoneNumberValidator to validate user information as the user enters it. Then, when the user clicks the Submit button to submit the form, you validate that the ZIP code is actually within the specified area code of the phone number.

You can also use the static `validateAll()` method to invoke all of the validators in an Array. This method returns an Array containing one ValidationResultEvent object for each validator that failed, and an empty Array if all validators succeed. The following example uses this method to invoke two validators in response to the `click` event for the Button control:

```
<?xml version="1.0"?>
<!-- validators\ValidatorMultipleValids.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initValidatorArray();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.validators.Validator;

            // Define the validator Array.
            private var myValidators:Array;
            private function initValidatorArray():void {
                myValidators=[zipV, pnV];
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:Model id="person">
            <userInfo>
                <zipCode>{zipCodeInput.text}</zipCode>
                <phoneNumber>{phoneNumberInput.text}</phoneNumber>
            </userInfo>
        </fx:Model>
        <!-- Define the validators. -->
        <mx:ZipCodeValidator id="zipV"
            source="{zipCodeInput}"
            property="text"/>
        <mx:PhoneNumberValidator id="pnV"
            source ="{phoneNumberInput}"
            property="text"/>
    </fx:Declarations>
    <s:Form>
        <s:FormItem>
            <s:TextInput id="zipCodeInput"/>
        </s:FormItem>
        <s:FormItem>
            <s:TextInput id="phoneNumberInput"/>
        </s:FormItem>
    </s:Form>
    <s:Button label="Validate"
        click="Validator.validateAll(myValidators);"/>
</s:Application>
```

Creating a reusable validator

You can define a reusable validator so that you can use it to validate multiple fields. To make it reusable, you programmatically set the `source` and `property` properties to specify the field to validate, or pass that information to the `validate()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ReusableVals.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;

            private function performValidation(eventObj:Event):void {
                zipV.listener=eventObj.currentTarget;
                zipV.validate(eventObj.currentTarget.text);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            triggerEvent=""/>
    </fx:Declarations>
    <s:TextInput id="shippingZip"
        focusOut="performValidation(event);"/>
    <s:TextInput id="billingZip"
        focusOut="performValidation(event);"/>
</s:Application>
```

In this example, you have two address areas for a customer: one for a billing address and one for a shipping address. Both addresses have a ZIP code field, so you can reuse a single ZipCodeValidator for both fields. The event listener for the `focusOut` event passes the field to validate to the `validate()` method.

Alternatively, you can write the `performValidation()` function as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ReusableValsSpecifySource.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;

            private function performValidation(eventObj:Event):void {
                zipV.source = eventObj.currentTarget;
                zipV.property = "text";
                zipV.validate();
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            triggerEvent=""/>
    </fx:Declarations>
    <s:TextInput id="shippingZip"
        focusOut="performValidation(event);"/>
    <s:TextInput id="billingZip"
        focusOut="performValidation(event);"/>
</s:Application>
```

Conditionalizing validator execution

By invoking a validator programmatically, you can use conditional logic in your application to determine which of several validators to invoke, set validator properties, or perform other preprocessing or postprocessing as part of the validation.

In the next example, you use a Button control to invoke a validator, but the application first determines which validator to execute, based on user input:

```
<?xml version="1.0"?>
<!-- validators\ConditionalVal.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.ValidationResultEvent;

            private var vEvent:ValidationResultEvent;

            private function validateData():void {
                if (String(country.selectedValue) == "Canada") {
                    vEvent = zipCN.validate(zipInput.text);
                }
                else {
                    vEvent = zipUS.validate(zipInput.text);
                }
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipUS"
            domain="US Only"
            listener="{zipInput}"/>
        <mx:ZipCodeValidator id="zipCN"
            domain="Canada Only"
            listener="{zipInput}"/>

        <s:RadioButtonGroup id="country"/>
    </fx:Declarations>
    <s:RadioButton group="{country}" label="US"/>
    <s:RadioButton group="{country}" label="Canada"/>
    <s:TextInput id="zipInput"/>
    <s:Button label="Submit" click="validateData();"/>
</s:Application>
```

In this example, you use a ZipCodeValidator to validate a ZIP code entered into a TextInput control. However, the `validateData()` function must first determine whether the ZIP code is for the U.S. or for Canada before performing the validation. In this example, the application uses the RadioButton controls to let the user specify the country as part of entering the ZIP code.

Validating required fields

All Flex validators contain a `required` property that, when set to `true`, causes validation to fail when a field is empty. You use this property to configure the validator to fail when a user does not enter data in a required input field.

You typically call the `validate()` method to invoke a validator on a required field. This is often necessary because you cannot guarantee that an event occurs to trigger the validation—an empty input field often means that the user never gave the input control focus.

The following example performs a validation on a required input field:

```
<?xml version="1.0"?>
<!-- validators\RequiredVal.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:StringValidator id="reqV"
            source="{inputA}"
            property="text"
            required="true"/>
    </fx:Declarations>
    <s:TextInput id="inputA"/>
    <s:Button label="Submit"
        click="reqV.validate();"/>
</s:Application>
```

In this example, the StringValidator executes when the following occurs:

- The TextInput control dispatches the `valueCommit` event. However, to dispatch that event, the user must give the TextInput control focus, and then remove focus. If the user never gives the TextInput control focus, the validator does not trigger, and Flex does not recognize that the control is empty. Therefore, you must call the `validate()` method to ensure that the validator checks for missing data.
- The user clicks the Button control. The validator issues a validation error when the user does not enter any data into the TextInput control. It also issues a validation error if the user enters an invalid String value.

Enabling and disabling a validator

The `enabled` property of a validator lets you enable and disable a validator. When the value of the `enabled` property is `true`, the validator is enabled; when the value is `false`, the validator is disabled. When a validator is disabled, it dispatches no events, and the `validate()` method returns `null`.

For example, you can set the `enabled` property by using data binding, as the following code shows:

```
<?xml version="1.0"?>
<!-- validators\EnableVal.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            protected function enableV_clickHandler(event:MouseEvent):void {
                // If there is an existing error message,
                // remove it when disabling the validator.
                if (enableV.selected == false)
                    inputA.errorString = "";
            }
        ]]>
    </fx:Script>


    <fx:Declarations>
        <mx:ZipCodeValidator id="zcVal"
            source="{inputA}"
            property="text"
            required="true"
            enabled="{enableV.selected}"/>
    </fx:Declarations>
    <s:TextInput id="inputA"/>
    <s:TextInput/>
    <s:CheckBox id="enableV"
        label="Validate input?"
        click="enableV_clickHandler(event)"/>
</s:Application>
```

In this example, you enable the validator only when the user selects the CheckBox control.

Using data binding to configure validators

You configure validators to match your application requirements. For example, the StringValidator lets you specify a minimum and maximum length of a valid string. For a String to be considered valid, it must be at least the minimum number of characters long, and less than or equal to the maximum number of characters.

Often, you set validator properties statically, which means that they do not change as your application executes. For example, the following StringValidator defines that the input string must be at least one character long and no longer than 10 characters:

```
<mx:StringValidator required="true" minlength="1" maxLength="10"/>
```

User input might also define the properties of the validator. In the following example, you let the user set the minimum and maximum values of a NumberValidator:

```
<?xml version="1.0"?>
<!-- validators\ConfigWithBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:NumberValidator
            source="{inputA}"
            property="text"
            minValue="{Number(inputMin.text)}"
            maxValue="{Number(inputMax.text)}"/>
    </fx:Declarations>
    <s:TextInput id="inputA"/>
    <s:TextInput id="inputMin" text="1"/>
    <s:TextInput id="inputMax" text="10"/>
</s:Application>
```

In this example, you use data binding to configure the properties of the validators.

General guidelines for validation

You should be aware of some guidelines when performing validation on forms. Typically, you associate forms with data models. That lets you trigger validation as part of binding an input user interface control to a field of the data model. You can also perform some of the following actions:

- If possible, assign validators to all the individual user-interface controls of the form. You can use a validator even if all that you want to do is to ensure that the user entered a value.
- Assign validators to multiple fields when necessary. For example, use the CreditCardValidator or the DateValidator with multiple fields.
- If you have any required fields, ensure that you explicitly call the `validate()` method on the validator. For more information, see “Validating required fields” on page 1981.
- Define a Submit button to invoke any validators before submitting data to a server. Typically, you use the `click` event of the Button control to invoke validators programmatically, and then submit the data if all validation succeeds.

The following example uses many of these guidelines to validate a form made up of several TextInput controls:

```
<?xml version="1.0"?>
<!-- validators\FullApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.ValidationResultEvent;
            private var vResult:ValidationResultEvent;
            // Function to validate data and submit it to the server.
            private function validateAndSubmit():void           {
                // Validate the required fields.
                vResult = fNameV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

                vResult = lNameV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;
                // Since the date requires 3 fields, perform the validation
                // when the Submit button is clicked.
                vResult = dayV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

                // Invoke any other validators or validation logic to make
                // an additional check before submitting the data.
                // Submit data to server.
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Define the data model. -->
        <fx:Model id="formInfo">
            <formData>
                <date>
                    <month>{monthInput.text}</month>
                    <day>{dayInput.text}</day>
                    <year>{yearInput.text}</year>
                </date>
                <name>
                    <firstName>{fNameInput.text}</firstName>
                    <lastName>{lNameInput.text}</lastName>
                </name>
                <phoneNum>{phoneInput.text}</phoneNum>
            </formData>
        </fx:Model>

        <!-- Define the validators. -->
        <mx:StringValidator id="fNameV"
            required="true"
            source="{fNameInput}"
            property="text"/>
        <mx:StringValidator id="lNameV"
```

```
            required="true"
            source="{lNameInput}"
            property="text"/>
        <mx:PhoneNumberValidator id="pnV"
            source="{phoneInput}"
            property="text"/>
        <!-- Invoke the DataValidator programmatically. -->
        <mx:DateValidator id="dayV"
            triggerEvent=""
            daySource="{dayInput}" dayProperty="text"
            monthSource="{monthInput}" monthProperty="text"
            yearSource="{yearInput}" yearProperty="text"/>
    </fx:Declarations>
    <!-- Define the form to populate the model. -->
    <s:Form>
        <s:FormItem label="Month">
            <s:TextInput id="monthInput"/>
        </s:FormItem>
        <s:FormItem label="Day">
            <s:TextInput id="dayInput"/>
        </s:FormItem>
        <s:FormItem label="Year">
            <s:TextInput id="yearInput"/>
        </s:FormItem>
        <s:FormItem label="First name">
            <s:TextInput id="fNameInput"/>
        </s:FormItem>
        <s:FormItem label="Last name">
            <s:TextInput id="lNameInput"/>
        </s:FormItem>
        <s:FormItem label="Phone">
            <s:TextInput id="phoneInput"/>
        </s:FormItem>
    </s:Form>
    <!-- Define the button to trigger validation. -->
    <s:Button label="Submit"
        click="validateAndSubmit();"/>
</s:Application>
```

In this example the following actions occur:

- The associated validator executes whenever the TextInput control dispatches a `valueCommit` event.
- The `click` event for the Button control invokes the `validateAndSubmit()` function to perform final validation before submitting data to the server.
- The `validateAndSubmit()` function invokes the validators for all required fields.
- The `validateAndSubmit()` function invokes the DateValidator because it requires three different input fields.
- Upon detecting the first validation error, the `validateAndSubmit()` function returns but does not submit the data.
- When all validations succeed, the `validateAndSubmit()` function submits the data to the server.

Working with validation errors

Subclasses of the UIComponent base class, which include the Flex user-interface components, generally handle validation failures by changing their border color and displaying an error message. When validation succeeds, components hide any existing validation error message and remove any border.

You can configure the content of the error messages and the display characteristics of validation errors.

Configuring error messages

All Flex validators define default error messages. In most cases, you can override these messages with your own.

The default error messages for all validators are defined by using resource bundles so that you can easily change them as part of localizing your application. You can override the default value of an error message for all validator objects created from a validator class by editing the resource bundles associated with that class.

You edit the error message for a specific validator object by writing a String value to a property of the validator. For example, the PhoneNumberValidator defines a default error message to indicate that an input phone number has the wrong number of digits. You can override the default error message for a specific PhoneNumberValidator object by assigning a new message string to the `wrongLengthError` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidatorErrMessage.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define the PhoneNumberValidator. -->
        <mx:PhoneNumberValidator id="pnV"
            source="{phoneInput}" property="text"
            wrongLengthError="Please enter a 10-digit number."/>
    </fx:Declarations>
    <!-- Define the TextInput control for entering the phone number. -->
    <s:TextInput id="phoneInput"/>
    <s:TextInput id="zipCodeInput"/>
</s:Application>
```

Changing the display of the validation error and error message

By default, the validation error message that appears when you move the mouse pointer over a user-interface control has a red background. You can use the `errorTip` style to change the color, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidatorErrMessageStyle.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <!-- Use blue for the error message. -->
    <fx:Style>
        .errorTip { borderColor: #0000FF}
    </fx:Style>
    <fx:Declarations>
        <!-- Define the PhoneNumberValidator. -->
        <mx:PhoneNumberValidator id="pnV"
            source="{phoneInput}" property="text"
            wrongLengthError="Please enter a 10-digit number."/>
    </fx:Declarations>
    <!-- Define the TextInput control for entering the phone number. -->
    <s:TextInput id="phoneInput"/>
    <s:TextInput id="zipCodeInput"/>
</s:Application>
```

In this example, the error message appears in blue.

For Spark components, when a validation error occurs, Flex uses the spark.skins.default.ErrorSkin skin class to draw a red box around the component associated with the failure. You can define a custom skin class to control the display of the validation error. If you define a custom skin class, set the `errorSkin` style of the Spark component that you are validating to the skin class.

Showing a validation error by using errorString

The UIComponent class defines the `errorString` property that you can use to show a validation error for a component, without actually using a validator class. When you write a String value to the `UIComponent.errorString` property, Flex draws a red border around the component to indicate the validation error, and the String appears in a ToolTip as the validation error message when you move the mouse over the component, just as if a validator detected a validation error.

To clear the validation error, write an empty String, " ", to the `UIComponent.errorString` property.

***Note:** Writing a value to the `UIComponent.errorString` property does not trigger the `valid` or `invalid` events; it only changes the border color and displays the validation error message.*

For information on writing custom ToolTip controls, see “ToolTip controls” on page 1933.

Clearing a validation error

The `errorString` property is useful when you want to reset a field that is a source for validation, and prevent a validation error from occurring when you reset the field.

For example, you might provide a form to gather user input. Within your form, you might also provide a button, or other mechanism, that lets the user reset the form. However, clearing form fields that are tied to validators could trigger a validation error. The following example uses the `errorString` property as part of resetting the `text` property of a TextInput control to prevent validation errors when the form resets:

```
<?xml version="1.0"?>
<!-- validators\ResetVal.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.events.ValidationResultEvent;
            private var vResult:ValidationResultEvent;

            // Function to validate data and submit it to the server.
            private function validateAndSubmit():void {
                // Validate the required fields.
                vResult = zipV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

                // Submit data to server.
            }
            // Clear the input controls and the errorString property
            // when resetting the form.
            private function resetForm():void {
                zipInput.text = '';
                zipInput.errorString = '';
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            source="{zipInput}"
            property="text"/>
    </fx:Declarations>
    <s:Form>
        <s:FormItem label="Enter ZIP code">
            <s:TextInput id="zipInput"/>
        </s:FormItem>
        <s:FormItem label="Enter Country">
            <s:TextInput id="cntryInput"/>
        </s:FormItem>
    </s:Form>
    <!-- Trigger submit. -->
    <s:Button label="Submit" click="validateAndSubmit();"/>

    <!-- Trigger reset. -->
    <s:Button label="Reset" click="resetForm();"/>
</s:Application>
```

In this example, the function that clears the form items also clears the `errorString` property associated with each item, clearing any validation errors.

Specifying a listener for validation

All validators support a listener property. When a validation occurs, Flex applies visual changes to the object specified by the `listener` property.

By default, Flex sets the `listener` property to the value of the source property. That means that all visual changes that occur to reflect a validation event occur on the component being validated. However, you might want to validate one component but have validation results apply to a different component, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\SetListener.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            source="{zipCodeInput}"
            property="text"
            listener="{errorMsg}"/>
    </fx:Declarations>
    <s:TextInput id="zipCodeInput"/>
    <s:TextArea id="errorMsg"/>
</s:Application>
```

Working with validation events

Flex gives you two ways to listen for validation events:

1 Listen for validation events dispatched by the component being validated.

Flex components dispatch `valid` and `invalid` events, depending on the results of the validation. This lets you listen for the events being dispatched from the component being validated, and perform any additional processing on the component based on its validation result.

The event object passed to the event listener is of type Event. For more information, including an example, see "Explicitly handling component validation events" on page 1990.

2 Listen for validation events dispatched by validators.

All validators dispatch `valid` or `invalid` events, depending on the results of the validation. You can listen for these events, and then perform any additional processing as required by your validator.

The event object passed to the event listener is of type ValidationResultEvent. For more information, including an example, see "Explicitly handing validator validation events" on page 1991.

You are not required to listen for validation events. When these events occur, by default, Flex changes the appropriate border color of the target component, displays an error message for an `invalid` event, or hides any previous error message for a `valid` event.

Explicitly handling component validation events

Sometimes you might want to perform some additional processing for a component if a validation fails or succeeds. In that case, you can handle the `valid` and `invalid` events yourself. The following example defines an event listener for the `invalid` event to perform additional processing when a validation fails:

```
<?xml version="1.0"?>
<!-- validators\ValCustomEventListener -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Import event class.
            import flash.events.Event;

            // Define vars for storing text colors.
            private var errorTextColor:Object = "red";
            private var currentTextColor:Object;

            // Initialization event handler for getting default text color.
            private function myCreationComplete(eventObj:Event):void {
                currentTextColor = getStyle('color');
            }

            // For an invalid event, change the text color.
            private function handleInvalidVal(eventObject:Event):void {
                setStyle('color', errorTextColor);
            }
            // For a valid event, restore the text color.
            private function handleValidVal(eventObject:Event):void {
                setStyle('color', currentTextColor);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:PhoneNumberValidator source="{phoneInput}" property="text"/>
    </fx:Declarations>
    <s:TextInput id="phoneInput"
        initialize="myCreationComplete(event);"
        invalid="handleInvalidVal(event);"
        valid="handleValidVal(event);"/>
    <s:TextInput id="zipInput"/>
</s:Application>
```

Explicitly handing validator validation events

To explicitly handle the `valid` and `invalid` events dispatched by validators, define an event listener, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValEventListener.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Import event class
            import mx.events.ValidationResultEvent;

            private function handleValid(event:ValidationResultEvent):void {
                if(event.type==ValidationResultEvent.VALID)
                    submitButton.enabled = true;
                else
                    submitButton.enabled = false;
            }
            // Submit form is everything is valid.
            private function submitForm():void {
                // Handle submit.
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:ZipCodeValidator
            source="{inputZip}" property="text"
            valid="handleValid(event);"
            invalid="handleValid(event);"/>
    </fx:Declarations>
    <s:TextInput id="inputZip"/>
    <s:TextInput id="inputPn"/>
    <s:Button id="submitButton"
        label="Submit"
        enabled="false"
        click="submitForm();"/>
</s:Application>
```

In this example, the Button control is disabled until the TextInput field contains a valid ZIP code. The `type` property of the event object is either `ValidationResultEvent.VALID` or `ValidationResultEvent.INVALID`, based on the result of the validation.

Within the event listener, you can use all the properties of the ValidationResultEvent class, including the following:

field A String that contains the name of the field that failed validation and triggered the event.

message A String that contains all the validator error messages created by the validation.

results An Array of ValidationResult objects, one for each field examined by the validator. For a successful validation, the `ValidationResultEvent.results` Array property is empty. For a validation failure, the `ValidationResultEvent.results` Array property contains one ValidationResult object for each field checked by the validator, both for fields that failed the validation and for fields that passed. Examine the `ValidationResult.isError` property to determine if the field passed or failed the validation.

Using standard validators

Flex includes the Validator subclasses. You use these validators for common types of data, including credit card numbers, dates, e-mail addresses, numbers, phone numbers, Social Security numbers, strings, and ZIP codes.

Validating credit card numbers

The CreditCardValidator class validates that a credit card number is the correct length, has the correct prefix, and passes the Luhn mod10 algorithm for the specified card type. This validator does not check whether the credit card is an actual active credit card account.

You typically use the `cardNumberSource` and `cardNumberProperty` properties to specify the location of the credit card number, and the `cardTypeSource` and `cardTypeProperty` properties to specify the location of the credit card type to validate.

The CreditCardValidator class validates that a credit card number is the correct length for the specified card type, as follows:

- Visa: 13 or 16 digits
- MasterCard: 16 digits
- Discover: 16 digits
- American Express: 15 digits
- DinersClub: 14 digits, or 16 digits if it also functions as a MasterCard

You specify the type of credit card number to validate by assigning a constant to the `cardTypeProperty` property. In MXML, valid constant values are:

- `"American Express"`
- `"Diners Club"`
- `"Discover"`
- `"MasterCard"`
- `"Visa"`

In ActionScript, you can use the following constants to set the `cardTypeProperty` property:

- `CreditCardValidatorCardType.AMERICAN_EXPRESS`
- `CreditCardValidatorCardType.DINERS_CLUB`
- `CreditCardValidatorCardType.DISCOVER`
- `CreditCardValidatorCardType.MASTER_CARD`
- `CreditCardValidatorCardType.VISA`

The following example validates a credit card number based on the card type that the users specifies. Any validation errors propagate to the Application object and open an Alert window.

```
<?xml version="1.0"?>
<!-- validators\CCExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Declarations>
        <mx:CreditCardValidator id="ccV"
            cardTypeSource="{cardTypeCombo.selectedItem}"
            cardTypeProperty="data"
            cardNumberSource="{cardNumberInput}"
            cardNumberProperty="text"/>
    </fx:Declarations>

    <s:Form id="creditCardForm">
        <s:FormItem label="Card Type">
            <s:ComboBox id="cardTypeCombo">
                <s:dataProvider>
                    <s:ArrayList>
                        <fx:Object label="American Express"
                            data="American Express"/>
                        <fx:Object label="Diners Club"
                            data="Diners Club"/>
                        <fx:Object label="Discover"
                            data="Discover"/>
                        <fx:Object label="MasterCard"
                            data="MasterCard"/>
                        <fx:Object label="Visa"
                            data="Visa"/>
                    </s:ArrayList>
                </s:dataProvider>
            </s:ComboBox>
        </s:FormItem>
        <s:FormItem label="Credit Card Number">
            <s:TextInput id="cardNumberInput"/>
        </s:FormItem>
        <s:FormItem>
            <s:Button label="Check Credit" click="ccV.validate();"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

The following example performs a similar validation, but uses the `source` and `property` properties to specify an object that contains the credit card information. In this example, you use the `listener` property to configure the validator to display validation error information on the TextInput control:

```
<?xml version="1.0"?>
<!-- validators\CCExampleSource.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var ccObj:Object = {cardType:String, cardNumber:String};

            public function valCC():void
            {
                // Populate ccObj with the data from the form.
                ccObj.cardType = cardTypeCombo.selectedItem.data;
                ccObj.cardNumber = cardNumberInput.text;
                // Validate ccObj.
                ccV.validate();
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:CreditCardValidator id="ccV"
            source="{this}"
            property="ccObj"
            listener="{cardNumberInput}"/>
    </fx:Declarations>

    <s:Form id="creditCardForm">
        <s:FormItem label="Card Type">
            <s:ComboBox id="cardTypeCombo">
                <s:dataProvider>
                    <s:ArrayList>
                        <fx:Object label="American Express"
                            data="American Express"/>
                        <fx:Object label="Diners Club"
                            data="Diners Club"/>
                        <fx:Object label="Discover"
                            data="Discover"/>
                        <fx:Object label="MasterCard"
                            data="MasterCard"/>
                        <fx:Object label="Visa"
                            data="Visa"/>
                    </s:ArrayList>
                </s:dataProvider>
            </s:ComboBox>
        </s:FormItem>
        <s:FormItem label="Credit Card Number">
            <s:TextInput id="cardNumberInput"/>
        </s:FormItem>
        <s:FormItem>
            <s:Button label="Check Credit" click="valCC();"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Validating currency

The Spark CurrencyValidator class checks that a string is a valid currency expression based on a set of parameters and the current `locale`. The CurrencyValidator class defines the properties that let you specify the format of the currency value, whether to allow negative values, and the precision of the values, and other options.

The digits of the input String can use national digits defined in the flash.globalization.NationalDigitsType class.

The following example uses the CurrencyValidator class to validate a currency value entered in U.S. dollars and in Euros:

```
<?xml version="1.0"?>
<!-- validators\SparkCurrencyExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="10" paddingLeft="10"/>
    </s:layout>

    <fx:Declarations>
        <!-- Example for US currency. -->
        <s:CurrencyValidator id="usV"
            locale="en-US"
            source="{priceUS}" property="text"
            trigger="{valButton}"
            triggerEvent="click"/>

        <!-- Example for European currency. -->
        <s:CurrencyValidator id="eurV"
            locale="fr-FR"
            source="{priceEU}" property="text"
            trigger="{valButton}"
            triggerEvent="click"/>
    </fx:Declarations>

    <s:Label text="Enter a US-formatted price:"/>
    <s:TextInput id="priceUS"/>

    <s:Label text="Enter a European-formatted price:"/>
    <s:TextInput id="priceEU"/>

    <s:Button id="valButton" label="Validate Currencies"/>
</s:Application>
```

In this example, you explicitly specify the `locale` for the two validators, rather than relying on the default. Use the first validator to validate currency values for a U.S. locale. Acceptable values include: $23.00, USD 23.00, or 123,456.00. Invalid values include: 2,37 EUR and €2.37.

The second validator specifies the `locale` as fr-FR for French currencies. Valid values include: 2,37, 2,37 EUR, and €2.37. Invalid values include: $23.00 and USD 23.00.

Validating dates

The DateValidator class validates that a String, Date, or Object contains a proper date and matches a specified format. Users can enter a single digit or two digits for month, day, and year. By default, the validator ensures that the following information is provided:

- The month is between 1 and 12 (or 0-11 for Date objects)
- The day is between 1 and 31
- The year is a number

If you specify a single String to validate, the String can contain digits and the formatting characters that the `allowedFormatChars` property specifies, including the slash (/), backslash (\), dash (-), and period (.) characters. By default, the input format of the date in a String is “mm/dd/yyyy” where “mm” is the month, “dd” is the day, and “yyyy” is the year. You can use the `inputFormat` property to specify a different format.

You can also specify to validate a date represented by a single Object, or by multiple fields of different objects. For example, you could use a data model that contains three fields that represent the day, month, and year portions of a date, or three TextInput controls that let a user enter a date as three separate fields. Even if you specify a date format that excludes a day, month, or year element, you must specify all three fields to the validator.

The following table describes how to specify the date to the DateValidator:

Validation source	Required properties	Default listener
String object containing the date	Use the `source` and `property` properties to specify the String.	Flex associates error messages with the field specified by the `property` property.
Date object containing the date	Use the `source` and `property` properties to specify the Date.	Flex associates error messages with the field specified by the `property` property.
Object or multiple fields containing the day, month, and year	Use all of the following properties to specify the day, month, and year inputs: `daySource`, `dayProperty`, `monthSource`, `monthProperty`, `yearSource`, and `yearProperty`.	Flex associates error messages with the field specified by the `daySource`, `monthSource`, and `yearSource` properties, depending on the field that caused the validation error.

The following example validates a date entered into a form:

```
<?xml version="1.0"?>
<!-- validators\DateExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:DateValidator id="dateV"
            daySource="{dayInput}" dayProperty="text"
            monthSource="{monthInput}" monthProperty="text"
            yearSource="{yearInput}" yearProperty="text"/>

        <!-- Alternate method for a single field containing the date. -->
        <fx:Model id="alternateDate">
            <dateInfo>
                <date>{dateInput.text}</date>
            </dateInfo>
        </fx:Model>
        <mx:DateValidator id="stringDateV"
            source="{dateInput}" property="text"
            inputFormat="dd/mm/yyyy"
            allowedFormatChars="*#~/"/>
    </fx:Declarations>
    <s:Form >
        <s:FormItem label="Month">
            <s:TextInput id="monthInput"/>
        </s:FormItem>
        <s:FormItem label="Day">
            <s:TextInput id="dayInput"/>
        </s:FormItem>
        <s:FormItem label="Year">
            <s:TextInput id="yearInput"/>
        </s:FormItem>
        <s:FormItem>
            <s:Button label="Check Date" click="dateV.validate();"/>
        </s:FormItem>
    </s:Form>
    <s:Form>
        <s:FormItem label="Date of Birth (dd/mm/yyyy)">
            <s:TextInput id="dateInput"/>
        </s:FormItem>
        <s:FormItem>
            <s:Button label="Check Date" click="stringDateV.validate();"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

In the next example, you validate a Date object:

```
<?xml version="1.0"?>
<!-- validators\DateObjectExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            // myDate is set to the current date and time.
            [Bindable]
            public var myDate:Date = new Date();
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:DateValidator id="dateV"
            source="{this}" property="myDate"
            valid="Alert.show('Validation Succeeded!');"/>
    </fx:Declarations>
    <s:Button label="Check Date" click="dateV.validate();"/>
</s:Application>
```

Validating e-mail addresses

The EmailValidator class validates that a string has an at sign character (@) and a period character (.) in the domain. You can use IP domain names if they are enclosed in square brackets; for example, myname@[206.132.22.1]. You can use individual IP numbers from 0 to 255. This validator does not check whether the domain and user name actually exist.

The following example validates an e-mail address to ensure that it is formatted correctly:

```
<?xml version="1.0"?>
<!-- validators\EmailExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Form id="contactForm">
        <s:FormItem id="homePhoneItem" label="Home Phone">
            <s:TextInput id="homePhoneInput"/>
        </s:FormItem>
        <s:FormItem id="cellPhoneItem" label="Cell Phone">
            <s:TextInput id="cellPhoneInput"/>
        </s:FormItem>
        <s:FormItem id="emailItem" label="Email">
            <s:TextInput id="emailInput"/>
        </s:FormItem>
    </s:Form>
    <fx:Declarations>
        <mx:PhoneNumberValidator id="pnVHome"
            source="{homePhoneInput}" property="text"/>
        <mx:PhoneNumberValidator id="pnVCell"
            source="{cellPhoneInput}" property="text"/>
        <mx:EmailValidator id="emV"
            source="{emailInput}" property="text"/>
    </fx:Declarations>
</s:Application>
```

Validating numbers

The Spark NumberValidator class ensures that a string represents a valid number for the current `locale`. This validator can ensure that the input falls within a given range, is an integer, is non-negative, does not exceed a specified precision, and other options. The NumberValidator also correctly validates formatted numbers (for example, "12,345.67").

The digits of the input String can use national digits defined in the flash.globalization.NationalDigitsType class.

The following example uses the NumberValidator class to ensure that an integer is between 1 and 10:

```
<?xml version="1.0"?>
<!-- validators\SparkNumberExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="700">

    <s:Form >
        <s:FormItem
            label="Number of Widgets (max 10 per customer)">
            <s:TextInput id="quantityInput"/>
        </s:FormItem>
        <s:FormItem >
            <s:Button label="Submit"/>
        </s:FormItem>
    </s:Form>

    <fx:Declarations>
        <s:NumberValidator id="numV"
            source="{quantityInput}" property="text"
            minValue="1" maxValue="10" domain="int"/>
    </fx:Declarations>
</s:Application>
```

Validating phone numbers

The PhoneNumberValidator class validates that a string is a valid phone number. A valid phone number contains at least 10 digits, plus additional formatting characters. This validator does not check if the phone number is an actual active phone number.

The following example uses two PhoneNumberValidator tags to ensure that the home and mobile phone numbers are entered correctly:

```
<?xml version="1.0"?>
<!-- validators\PhoneExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Form id="contactForm">
        <s:FormItem id="homePhoneItem" label="Home Phone">
            <s:TextInput id="homePhoneInput"/>
        </s:FormItem>
        <s:FormItem id="cellPhoneItem" label="Cell Phone">
            <s:TextInput id="cellPhoneInput"/>
        </s:FormItem>
        <s:FormItem id="emailItem" label="Email">
            <s:TextInput id="emailInput"/>
        </s:FormItem>
    </s:Form>
    <fx:Declarations>
        <mx:PhoneNumberValidator id="pnVHome"
            source="{homePhoneInput}" property="text"/>
        <mx:PhoneNumberValidator id="pnVCell"
            source="{cellPhoneInput}" property="text"/>
        <mx:EmailValidator id="emV"
            source="{emailInput}" property="text"/>
    </fx:Declarations>
</s:Application>
```

Validating using regular expressions

The RegExpValidator class lets you use a regular expression to validate a field. You pass a regular expression to the validator by using the `expression` property, and additional flags to control the regular expression pattern matching by using the `flags` property.

The validation is successful if the validator can find a match of the regular expression in the field to validate. A validation error occurs when the validator finds no match.

You use regular expressions with the RegExpValidator. For information on writing regular expressions, see ActionScript 3.0 Developer's Guide.

The RegExpValidator class dispatches the `valid` and `invalid` events. For an `invalid` event, the event object is an instance of the ValidationResultEvent class, and it contains an Array of ValidationResult objects.

However, for a `valid` event, the ValidationResultEvent object contains an Array of RegExpValidationResult objects. The RegExpValidationResult class is a child class of the ValidationResult class, and contains additional properties that you use with regular expressions, including the following:

matchedIndex An integer that contains the starting index in the input String of the match.

matchedString A String that contains the substring of the input String that matches the regular expression.

matchedSubStrings An Array of Strings that contains parenthesized substring matches, if any. If no substring matches are found, this Array is of length 0. Use `matchedSubStrings[0]` to access the first substring match.

The following example uses the regular expression ABC\d to cause the validator to match a pattern consisting of the letters A, B, and C in sequence followed by any digit:

```
<?xml version="1.0"?>
<!-- validators\RegExpExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
          import mx.events.ValidationResultEvent;
          import mx.validators.*;

          private function handleResult(event:ValidationResultEvent):void {
              if (event.type == "valid")
              {
                  // For valid events, the results Array contains
                  // RegExpValidationResult objects.
                  var xResult:RegExpValidationResult;
                  myTA.text="";
                  for (var i:uint = 0; i < event.results.length; i++)
                  {
                      xResult = event.results[i];
                      myTA.text=myTA.text + xResult.matchedIndex + " " +
                          xResult.matchedString + "\n";
                  }
              }
              else
              {
                  // Not necessary, but if you needed to access it,
                  // the results array contains ValidationResult objects.
                  var result:ValidationResult;
                  myTA.text="";
              }
          }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <mx:RegExpValidator id="regExpV"
            source="{exp}" property="text"
            flags="g"
            expression="{source.text}"
            valid="handleResult(event);"
            invalid="handleResult(event);"/>
    </fx:Declarations>

    <s:Form>
        <s:FormItem label="Search string">
            <s:TextInput id="exp"/>
        </s:FormItem>
        <s:FormItem label="Regular expression">
            <s:TextInput id="source" text="ABC\d"/>
        </s:FormItem>
        <s:FormItem label="Results">
            <s:TextArea id="myTA"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

In this example, you specify the regular expression in the TextInput control named source, and bind it to the `expression` property of the validator. You can modify the regular expression by entering a new expression in the TextInput control. A value of `g` for the `flags` property specifies to find multiple matches in the input field.

The event handler for the `valid` event writes to the TextArea control the index in the input String and matching substring of all matches of the regular expression. The `invalid` event handler clears the TextArea control.

Validating social security numbers

The SocialSecurityValidator class validates that a string is a valid United States Social Security Number. This validator does not check if the number is an existing Social Security Number.

The following example validates a Social Security Number:

```
<?xml version="1.0"?>
<!-- validators\SSExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Form id="identityForm">
        <s:FormItem id="ssnItem" label="Social Security Number">
            <s:TextInput id="ssnField"/>
        </s:FormItem>
        <s:FormItem id="licenseItem" label="Driver's License Number">
            <s:TextInput id="licenseInput"/> <!-- Not validated -->
        </s:FormItem>
    </s:Form>
    <fx:Declarations>
        <mx:SocialSecurityValidator id="ssV"
            source="{ssnField}" property="text"/>
    </fx:Declarations>
</s:Application>
```

Validating strings

The StringValidator class validates that a string length is within a specified range. The following example ensures that a string is between 6 and 12 characters long:

```
<?xml version="1.0"?>
<!-- validators\StringExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Form id="membershipForm">
        <s:FormItem id="fullNameItem" label="Full Name">
        <!-- Not validated -->
            <s:TextInput id="fullNameInput"/>
        </s:FormItem>
        <s:FormItem id="userNameItem" label="Username">
            <s:TextInput id="userNameInput"/>
        </s:FormItem>
    </s:Form>
    <fx:Declarations>
        <mx:StringValidator source="{userNameInput}" property="text"
            minLength="6" maxLength="12"/>
    </fx:Declarations>
</s:Application>
```

Validating ZIP codes

The ZipCodeValidator class validates that a string has the correct length for a five-digit ZIP code, a five-digit+four-digit United States ZIP code, or a Canadian postal code.

The following example validates either a United States ZIP code or a Canadian postal code:

```
<?xml version="1.0"?>
<!-- validators\ZCExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Form id="addressForm">
        <s:FormItem id="zipCodeItem" label="Zip Code">
            <s:TextInput id="zipInput"/>
        </s:FormItem>
        <s:FormItem id="submitArea">
            <s:Button label="Submit"/>
        </s:FormItem>
    </s:Form>
    <fx:Declarations>
        <mx:ZipCodeValidator id="zipV"
            source="{zipInput}" property="text"
            domain="US or Canada"/>
    </fx:Declarations>
</s:Application>
```

Formatting Data

Data formatters are user-configurable objects that format raw data into a customized string. You often use formatters with data binding to create a meaningful display of the raw data bound to a component. This can save you time by automating data formatting tasks and by letting you easily change the formatting of fields within your applications.

Using formatters

Formatters in Adobe® Flex® are components that you use to format data into strings. Formatters perform a one-way conversion of raw data to a formatted string. You typically trigger a formatter just before displaying data in a text field. Flex includes standard formatters that let you format currency, dates, numbers, phone numbers, and ZIP codes.

Spark and MX formatters

Flex defines two sets of formatters: Spark and MX. The Spark formatters rely on the classes in the flash.globalization package. The flash.globalization classes use the locale data provided by the operating system. Therefore, Spark formatters provide behavior that is consistent with the operating system and have access to all the locales that are supported by the operating system.

MX formatters use the Flex ResourceManager to access locale-specific data from properties files that are included in the Flex SDK. The MX formatters provide the same behavior across operating systems, but are limited to the locales provided by the Flex SDK or by the application developer.

The following table lists the Spark and MX formatting classes. When possible, Adobe recommends that you use the Spark formatters in your application:

Spark formatter	**MX formatter**	**Description**
CurrencyFormatter	CurrencyFormatter	Format a currency value.
DataTimeFormatter	DateTimeFormatter	Format a date and time value.
NumberFormatter	NumberFormatter	Format a numeric value.
	PhoneFormatter	Format a phone number.
	SwitchSymbolFormatter	Format a String by replacing placeholder characters in one String with numbers from a second String.
	ZipCodeFormatter	Format a U.S. or Canadian postal code.

Formatter example

The following steps describe the general process for using a formatter:

1 Declare a formatter in an `<fx:Declarations>` tag in your MXML code, specifying the appropriate formatting properties. You define formatters in an `<fx:Declarations>` tag because they are not visual components.

You can also create formatters in ActionScript for use in an application.

2 Call the formatter's `format()` method, and specify the value to be formatted as a parameter to the `format()` method. The `format()` method returns a String containing the formatted value.

The following example formats a phone number that a user inputs in an application by using the TextInput control:

```
<?xml version="1.0"?>
<!-- formatters\Formatter2TextFields.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Declare a PhoneFormatter and define formatting parameters.-->
        <mx:PhoneFormatter id="phoneDisplay"
            areaCode="415"
            formatString="###-####" />
    </fx:Declarations>
    <!-- Declare the input control.-->
    <s:Label text="Enter 7 digit phone number (########):"/>
    <s:TextInput id="myTI"/>

    <!-- Declare the control to display the formatted data.-->
    <s:TextArea text="{phoneDisplay.format(myTI.text)}"/>
</s:Application>
```

In this example, you use the MX PhoneFormatter class to display the formatted phone number in a TextArea control. The `format()` method is called as part of a data binding expression when the `text` property of the TextInput control changes.

You can call the `format()` method from anywhere in your application, typically in response to an event. The following example declares a Spark DateTimeFormatter. The DateTimeFormatter uses the default locale-specific format for the input String. The application then writes a formatted date to the text property of a TextInput control in response to the `click` event of a Button control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- formatters\sparkformatters\SparkFormatterDateField.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
    </s:layout>

    <fx:Declarations>
        <!-- Declare a Spark DateTimeFormatter and
             define formatting parameters.-->
        <s:DateTimeFormatter id="dateTimeFormatter"/>
    </fx:Declarations>

    <s:Label text="Enter date (mm/dd/yyyy):"/>
    <s:TextInput id="dob" text=""/>

    <s:Label text="Formatted date: "/>
    <s:TextInput id="formattedDate"
        text="" width="300"
        editable="false"/>

    <!-- Format and update the date.-->
    <s:Button label="Format Input"
        click="formattedDate.text=dateTimeFormatter.format(dob.text);"/>
</s:Application>
```

Spark formatters

Flex includes the following Spark classes for formatting dates, numbers and currencies:

- spark.formatters.CurrencyFormatter
- spark.formatters.NumberFormatter
- spark.formatters.DateTimeFormatter

The Spark formatters provide the following functionality:

- Locale-specific formatting of dates, times, number, and currency amounts.
- Locale-specific parsing of numbers and currency amounts.
- Locale-specific string comparison. For more information, see “Sorting and matching” on page 2062.
- Locale-specific uppercase and lowercase string conversion.

The Spark formatters use the `locale` style property to determine how to format data based on the locale. The `locale` style is an inheritable style that you can set for the entire application or specifically for a particular formatter.

If a Spark formatter does not explicitly set the `locale` style, then it uses the value specified by the `locale` style of the application container. If you do not set the `locale` style property, the application uses the global default from the defaults.css style sheet, which defaults to `en`. For more information, see “Setting the locale” on page 2056.

You can explicitly configure the formatter to use the default locale by setting the `locale` style to the constant value `flash.globalization.LocaleID.DEFAULT`.

Adobe Product Evangelist James Ward provides an overview of the Spark formatters in the video Overview of Spark Formatters in Flex.

See a video about Spark formatters from video2brain at New Formatters and Validators.

More Help topics

Adobe TV: Spark Formatters

“Setting the locale” on page 2056

“Formatting dates, numbers, and currencies” on page 2057

Error handling in a Spark formatter

You call the `format()` method to format a value. By default, if the `format()` method fails, it returns null. However, you can use the `errorText` property of the formatter to define a String to return, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- formatters\sparkformatters\SparkFormatterError.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
    </s:layout>

    <fx:Declarations>
        <!-- Declare a Spark DateTimeFormatter and
        define formatting parameters.-->
        <s:DateTimeFormatter id="dateTimeFormatter"
            dateTimePattern="month: MM, day: DD, year: YYYY"
            errorText="Invalid input value"/>
    </fx:Declarations>

    <s:Label text="Enter date (mm/dd/yyyy):"/>
    <s:TextInput id="dob" text=""/>

    <s:Label text="Formatted date: "/>
    <s:TextInput id="formattedDate"
        text="" width="300"
        editable="false"/>

    <!-- Format and update the date.-->
    <s:Button label="Format Input"
        click="formattedDate.text=dateTimeFormatter.format(dob.text);"/>
</s:Application>
```

Formatting numbers with a Spark formatter

The Spark NumberFormatter class provides basic formatting options for numeric data, including decimal formatting, thousands separator formatting, and negative sign formatting.

The format() method accepts a number or a number formatted as a String value, and returns the formatted string. When a number formatted as a String value is passed to the format() method, the function first uses the Number() function to convert the String to a Number object.

The following example uses the NumberFormatter class in an MXML file:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">
    <s:layout>
        <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            protected function button2_clickHandler(event:MouseEvent):void {
                displayVal.text = PrepForDisplay.format(bigNumber.text);
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <!-- Declare and define parameters for the NumberFormatter.-->
        <s:NumberFormatter id="PrepForDisplay"
            fractionalDigits="3"
            decimalSeparator="."
            groupingSeparator=","
            useGrouping="true"
            negativeNumberFormat="0"
            errorText="'{bigNumber.text}' is an invalid input value"/>
    </fx:Declarations>

    <!-- String to format.-->
    <s:TextInput id="bigNumber" text="Enter number"/>
    <s:Button label="Format Value"
        click="button2_clickHandler(event);"/>

    <!-- Display formatted value.-->
    <s:Label id="displayVal"/>
</s:Application>
```

Formatting currency with a Spark formatter

The Spark CurrencyFormatter class provides the same features as the NumberFormatter class, plus a currency symbol. It also has additional properties to control the formatted result: currencySymbol, negativeCurrencyFormat, and positiveCurrencyFormat.

The format() method accepts a Number value or a number formatted as a String value and formats the resulting string. When a number formatted as a String value is passed to the format() method, the function first uses the Number() function to convert the String to a Number object.

The useCurrencySymbol property determines the currency symbol returned in the formatted String based on the locale style property. When the useCurrencySymbol property is set to true, the formatter returns the value of the currencySymbol property in the formatted String. When the useCurrencySymbol property is set to false, the formatter returns the value of the currencyISOCode property in the formatted String.

The following example uses the CurrencyFormatter class in an MXML file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- formatters\sparkformatters\SparkCurrencyFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            protected function button1_clickHandler(event:MouseEvent):void {
                displayVal.text = Price.format(currVal.text);
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <!-- Declare a CurrencyFormatter and define parameters.-->
        <s:CurrencyFormatter id="Price"
            currencySymbol="$"
            useCurrencySymbol="true"
            negativeCurrencyFormat="0"
            positiveCurrencyFormat="0"
            errorText="'{currVal.text}' is an invalid input value"/>
    </fx:Declarations>

    <!-- Enter currency value, then click the Button to format. -->
    <s:TextInput id="currVal" text="Enter value"/>
    <s:Button label="Format Value"
        click="button1_clickHandler(event);"/>
    <!-- Display formatted value.-->
    <s:Label id="displayVal"/>

</s:Application>
```

Formatting dates with a Spark formatter

The Spark DateTimeFormatter class gives you a wide range of combinations for displaying date and time information. The `format()` method accepts a Date object, which it converts to a String based on a user-defined pattern.

The `format()` method can also accept a String-formatted date. If you pass a String, the formatter first converts it to a Date object by using the Date constructor. The hour value in the String must be between 0 and 23, inclusive. The minutes and seconds value must be between 0 and 59, inclusive. If the `format()` method is unable to parse the String into a Date object, it returns `null`.

The following examples show possible input strings to the `format()` method:

```
"12/31/98" or "12-31-98" or "1998-12-31" or "12/31/1998"
"Friday, December 26, 2005 8:35 am"
"Jan. 23, 1989 11:32:25"
```

Using pattern strings

Use the `dateTimePattern` property of the DateTimeFormatter class to specify the string of pattern letters that determine the appropriate formatting. You compose a pattern string by using specific letter sequences. For example, “yyyy/MM”.

***Note:** By omitting a pattern string, the date is formatted using the default for the current locale. By specifying a pattern string, you override the default formatting. Information, such as month and day names, are still displayed correctly for the locale, but the order of elements in the formatted string is fixed by the specified pattern string.*

You can include text and punctuation in the pattern string. However the characters from “a” to “z” and from “A” to “Z” are reserved as syntax characters. Enclose these characters in single quotes to include them in the formatted output string. To include a single quote, use two single quotes. The two single quotes may appear inside or outside a quoted portion of the pattern string.

When you create a pattern string, you usually repeat pattern letters. The number of repeated letters determines the presentation. For numeric values, the number of pattern letters is the minimum number of digits; shorter numbers are zero-padded to this amount. For example, the four-letter pattern string “yyyy” corresponds to a four-digit year value.

In cases where there is a corresponding mapping of a text description—for example, the name of a month—the full form is used if the number of pattern letters is four or more; otherwise, a short or abbreviated form is used, if available. For example, if you specify the pattern string “MMMM” for month, the formatter requires the full month name, such as “December”, instead of the abbreviated month name, such as “Dec”.

For time values, a single pattern letter is interpreted as one or two digits. Two pattern letters are interpreted as two digits.

The complete list of options to the pattern string is defined by the setDateTimePattern() method of the flash.globalization.DateTimeFormatter class.

The following table shows sample pattern strings and the resulting presentation:

Pattern	Result
`EEEE, MMMM dd, yyyy h:mm:ss a` (default)	Wednesday, December 01, 2010 3:06:43 PM
`yyyy.MM.dd 'at' h:mm:ss`	2010.12.01 at 3:19:44
`EEE, MMM d, yy`	Wed, Dec 1, 10
`kk 'o''clock' a`	15 o'clock PM
`k:mm a`	12:08 PM
`K:mm a`	0:08 PM
`yyyy.MMMM.dd HH:mm:ss`	2005.July.04 12:08:34

Example: Using the Spark DateFormatter class

The following example uses the DateTimeFormatter class in an MXML file for formatting a Date object as a String:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- formatters\spark\SparkDateFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">
    <s:layout>
        <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            // Define variable to hold the date.
            [Bindable]
            private var today:Date = new Date();
        ]]>
    </fx:Script>

    <fx:Declarations>
        <!-- Declare a DateFormatter and define parameters.-->
        <s:DateTimeFormatter id="DateDisplay1"
            dateTimePattern="MMMM d, yyyy"/>
        <s:DateTimeFormatter id="DateDisplay2"
            dateTimePattern="hh:mm a"/>
        <s:DateTimeFormatter id="DateDisplay3"
            dateTimePattern="hh:mm a, MMMM d, yyyy"/>
        <s:DateTimeFormatter id="DateDisplay4"
            dateTimePattern="yyyy.MMMM.dd HH:mm:ss"/>
</fx:Declarations>

    <!-- Display the date in a Label control.-->
    <s:Label id="myTA1" text="{DateDisplay1.format(today)}"/>
    <!-- Display the date in a Label control.-->
    <s:Label id="myTA2" text="{DateDisplay2.format(today)}"/>
    <!-- Display the date in a Label control.-->
    <s:Label id="myTA3" text="{DateDisplay3.format(today)}"/>
    <!-- Display the date in a Label control.-->
    <s:Label id="myTA4" text="{DateDisplay4.format(today)}"/>
</s:Application>
```

MX formatters

All formatters are subclasses of the mx.formatters.Formatter class. The Formatter class declares a format() method that takes a value and returns a String value.

Writing an error handler function for an MX formatter

For most formatters, when an error occurs, the `format()` method returns an empty string is and a string describing the error is written to the formatter's `error` property. You can check for an empty string in the return value of the `format()` method, and if found, access the `error` property to determine the cause of the error.

Alternatively, you can write an error handler function that returns an error message for a formatting error. The following example shows a simple error handler function:

```
<?xml version="1.0"?>
<!-- formatters\FormatterSimpleErrorForDevApps.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            private function formatWithError(value:Object):String {
                var formatted:String = myFormatter.format(value);
                if (formatted == "") {
                    if (myFormatter.error != null ) {
                        if (myFormatter.error == "Invalid value") {
                            formatted = ": The value is not valid.";
                        }
                        else {
                            formatted = ": The formatString is not valid.";
                        }
                    }
                }
                return formatted;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Declare a formatter and specify formatting properties.-->
        <mx:DateFormatter id="myFormatter"
            formatString="MXXXMXMXMXMXM"/>
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data.-->
    <s:TextInput id="myTI"
        width="75%"
        text="Your order shipped on {formatWithError('May 23, 2005')}"/>
</s:Application>
```

In this example, you define a DateFormatter with an invalid format string. You then use the `formatWithError()` method to invoke the formatter in the TextInput control, rather than calling the Date formatter’s `format()` method directly. In this example, if either the input string or the format string is invalid, the `formatWithError()` method returns an error message instead of a formatted String value.

Formatting currency with an MX formatter

The MX CurrencyFormatter class provides the same features as the NumberFormatter class, plus a currency symbol. It has two additional properties: `currencySymbol` and `alignSymbol`. (For more information about the NumberFormatter class, see “Formatting numbers with an MX formatter” on page 2017.)

The CurrencyFormatter class provides basic formatting options for numeric data, including decimal formatting, thousands separator formatting, and negative sign formatting. The `format()` method accepts a Number value or a number formatted as a String value and formats the resulting string.

When a number formatted as a String is passed to the `format()` method, the function parses the string from left to right and attempts to extract the first sequence of numbers it encounters. The function parses the thousands separators and decimal separators along with their trailing numbers. The parser searches for a comma (,) for the thousands separator unless you set a different character in the `thousandsSeparatorFrom` property. The parser searches for a period (.) for the decimal separator unless you define a different character in the `decimalSeparator` property.

***Note:** When a number is provided to the `format()` method as a String value, a negative sign is recognized if it is a dash (-) immediately preceding the first number in the sequence. A dash, space, and then a first number are not interpreted as a negative sign.*

Example: Using the MX CurrencyFormatter class

The following example uses the CurrencyFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainCurrencyFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Define variable to hold the price.
            [Bindable]
            private var todaysPrice:Number=4025;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Declare a CurrencyFormatter and define parameters.-->
        <mx:CurrencyFormatter id="Price" precision="2"
            rounding="none"
            decimalSeparatorTo="."
            thousandsSeparatorTo=","
            useThousandsSeparator="true"
            useNegativeSign="true"
            currencySymbol="$"
            alignSymbol="left"/>
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data.-->
    <s:TextInput text="Today's price is {Price.format(todaysPrice)}."
        width="200"/>
</s:Application>
```

At run time, the following text is displayed:

Today’s price is $4,025.00.

Error handling: MX CurrencyFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the error property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String.
`Invalid format`	One or more of the parameters contain an unusable setting.

Formatting dates with an MX formatter

The MX DateFormatter class gives you a wide range of combinations for displaying date and time information. The `format()` method accepts a Date object, which it converts to a String based on a user-defined pattern. The `format()` method can also accept a String-formatted date, which it attempts to parse into a valid Date object prior to formatting.

The DateFormatter class has a `parseDateString()` method that accepts a date formatted as a String. The `parseDateString()` method examines sections of numbers and letters in the String to build a Date object. The parser is capable of interpreting long or abbreviated (three-character) month names, time, am and pm, and various representations of the date.

The hour value in the String must be between 0 and 23, inclusive. The minutes and seconds value must be between 0 and 59, inclusive. If the `parseDateString()` method is unable to parse the String into a Date object, it returns `null`.

The following examples show some of the ways strings can be parsed:

```
"12/31/98" or "12-31-98" or "1998-12-31" or "12/31/1998"
"Friday, December 26, 2005 8:35 am"
"Jan. 23, 1989 11:32:25"
```

The DateFormatter class parses strings from left to right. A date value should appear before the time and must be included. The time value is optional. A time signature of 0:0:0 is the Date object’s default for dates that are defined without a time. Time zone offsets are not parsed.

Using Pattern strings

You provide the DateFormatter class with a string of pattern letters, which it parses to determine the appropriate formatting. You must understand how to compose the string of pattern letters to control the formatting options and the format of the string that is returned.

You compose a pattern string by using specific uppercase letters: for example, YYYY/MM. The DateFormatter pattern string can contain other text in addition to pattern letters. To form a valid pattern string, you need only one pattern letter.

When you create a pattern string, you usually repeat pattern letters. The number of repeated letters determines the presentation. For numeric values, the number of pattern letters is the minimum number of digits; shorter numbers are zero-padded to this amount. For example, the four-letter pattern string “YYYY” corresponds to a four-digit year value.

In cases where there is a corresponding mapping of a text description—for instance, the name of a month—the full form is used if the number of pattern letters is four or more; otherwise, a short or abbreviated form is used, if available. For example, if you specify the pattern string “MMMM” for month, the formatter requires the full month name, such as “December”, instead of the abbreviated month name, such as “Dec”.

For time values, a single pattern letter is interpreted as one or two digits. Two pattern letters are interpreted as two digits.

The following table describes each of the available pattern letters:

Pattern letter	**Description**

Y	Year. If the number of pattern letters is two, the year is truncated to two digits; otherwise, it appears as four digits. The year can be zero-padded, as the third example shows in the following set of examples:
	Examples:
	YY = 05
	YYYY = 2005
	YYYYY = 02005

M	Month in year. The format depends on the following criteria: • If the number of pattern letters is one, the format is interpreted as numeric in one or two digits. • If the number of pattern letters is two, the format is interpreted as numeric in two digits. • If the number of pattern letters is three, the format is interpreted as short text. • If the number of pattern letters is four, the format is interpreted as full text. Examples: M = 7 MM= 07 MMM=Jul MMMM= July

D	Day in month. While a single-letter pattern string for day is valid, you typically use a two-letter pattern string.
	Examples:
	D=4
	DD=04
	DD=10

E	Day in week. The format depends on the following criteria: • If the number of pattern letters is one, the format is interpreted as numeric in one or two digits. • If the number of pattern letters is two, the format is interpreted as numeric in two digits. • If the number of pattern letters is three, the format is interpreted as short text. • If the number of pattern letters is four, the format is interpreted as full text. Examples: E = 1 EE = 01 EEE = Mon EEEE = Monday
A	am/pm indicator.
J	Hour in day (0-23).
H	Hour in day (1-24).

Pattern letter	Description
K	Hour in am/pm (0-11).
L	Hour in am/pm (1-12).

N	Minute in hour.
	Examples:
	N = 3
	NN = 03

S	Second in minute. Examples: SS = 30
Q	Millisecond in seconds.Example:s: QQ = 78QQQ = 078
Other text	You can add other text into the pattern string to further format the string. You can use punctuation, numbers, and all lowercase letters. You should avoid uppercase letters because they may be interpreted as pattern letters. Example: EEEE, MMM. D, YYYY at L:NN A = Tuesday, Sept. 8, 2005 at 1:26 PM

The following table shows sample pattern strings and the resulting presentation:

Pattern	Result
`YYYY.MM.DD at HH:NN:SS`	2005.07.04 at 12:08:56
`EEEE, MMM. D, YYYY at L:NN:QQQ A`	Tuesday, Sept. 8, 2005 at 1:26:012 PM
`EEE, MMM D, 'YY`	Wed, Jul 4, '05
`H:NN A`	12:08 PM
`HH o'clock A`	12 o'clock PM
`K:NN A`	0:08 PM
`YYYYY.MMMM.DD. JJ:NN A`	02005.July.04. 12:08 PM
`EEE, D MMM YYYY HH:NN:SS`	Wed, 4 Jul 2005 12:08:56

Example: Using the DateFormatter class

The following example uses the DateFormatter class in an MXML file for formatting a Date object as a String:

```
<?xml version="1.0"?>
<!-- formatters\MainDateFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Define variable to hold the date.
            [Bindable]
            private var today:Date = new Date();
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Declare a DateFormatter and define parameters.-->
        <mx:DateFormatter id="DateDisplay"
            formatString="MMMM D, YYYY"/>
    </fx:Declarations>
    <!-- Display the date in a TextArea control.-->
    <s:TextInput id="myTA" text="{DateDisplay.format(today)}"/>
</s:Application>
```

At run time, the TextInput control displays the current date.

Error handling: DateFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the error property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	A value that is not a Date object is passed to the `format()` method. (An empty argument is allowed.)
`Invalid format`	• The `formatString` property is set to empty (""). • There is less than one pattern letter in the `formatString` property.

Formatting numbers with an MX formatter

The MX NumberFormatter class provides basic formatting options for numeric data, including decimal formatting, thousands separator formatting, and negative sign formatting. The `format()` method accepts a number or a number formatted as a String value, and formats the resulting string.

When a number formatted as a String value is passed to the `format()` method, the function parses the string from left to right and attempts to extract the first sequence of numbers it encounters. The function parses the thousands separators and decimal separators along with their trailing numbers. The parser searches for a comma (,) for the thousands separator unless you set a different character in the `thousandsSeparatorFrom` property. The parser searches for a period (.) for the decimal separator unless you define a different character in the `decimalSeparator` property.

Note: *The `format()` method recognizes a dash (-) immediately preceding the first number in the sequence as a negative number. A dash, space, and then number sequence are not interpreted as a negative number.*

The `rounding` and `precision` properties of the MX NumberFormatter class affect the formatting of the decimal in a number. If you use both `rounding` and `precision` properties, rounding is applied first, and then the decimal length is set by using the specified precision value. This lets you round a number and still have a trailing decimal; for example, 303.99 = 304.00.

Example: Using the MX NumberFormatter class

The following example uses the MX NumberFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainNumberFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Define variable to hold the number.
            [Bindable]
            private var bigNumber:Number = 6000000000.65;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Declare and define parameters for the NumberFormatter.-->
        <mx:NumberFormatter id="PrepForDisplay"
            precision="0"
            rounding="up"
            decimalSeparatorTo="."
            thousandsSeparatorTo=","
            useThousandsSeparator="true"
            useNegativeSign="true"/>
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data.-->
    <s:TextInput text="{PrepForDisplay.format(bigNumber)}"/>
</s:Application>
```

At run time, the following text appears:

6,000,000,001

Error handling: MX NumberFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the error property. An error refers to a problem with the value being submitted or the format string that contains the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String.
`Invalid format`	One or more of the parameters contain an unusable setting.

Formatting phone numbers with an MX formatter

The PhoneFormatter class lets you format a phone number by adjusting the format of the area code and the subscriber code. You can also adjust the country code and configuration for international formats. The value passed into the `PhoneFormatter.format()` method must be a Number object or a String object with only digits.

The PhoneFormatter `formatString` property accepts a formatted string as a definition of the format pattern. The following table shows common options for `formatString` values. The `format()` method for the PhoneFormatter accepts a sequence of numbers. The numbers correspond to the number of placeholder (#) symbols in the `formatString` value. The number of placeholder symbols in the `formatString` property and the number of digits in the `format()` method value must match.

formatString value	Input	Output
`###-####`	1234567	(xxx) 456-7890
`(###) ### ####`	1234567890	(123) 456-7890
`###-###-####`	11234567890	123-456-7890
`#(###) ### ####`	11234567890	1(123) 456 7890
`#-###-###-####`	11234567890	1-123-456-7890
`+###-###-###-####`	1231234567890	+123-123-456-7890

In the preceding table, dashes (-) are used as separator elements where applicable. You can substitute period (.) characters or blank spaces for the dashes. You can change the default allowable character set as needed by using the `validPatternChars` property. You can change the default character that represents a numeric placeholder by using the `numberSymbol` property (for example, to change from # to $).

***Note:** A shortcut is provided for the United States seven-digit format. If the `areaCode` property contains a value and you use the seven-digit format string, a seven-digit format entry automatically adds the area code to the string returned. The default format for the area code is (###). You can change this by using the `areaCodeFormat` property. You can format the area code any way you want as long as it contains three number placeholders.*

Example: Using the PhoneFormatter class

The following example uses the PhoneFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainPhoneFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Define variable to hold the phone number.
            [Bindable]
            private var newNumber:Number = 1234567;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Declare a PhoneFormatter and define formatting parameters.-->
        <mx:PhoneFormatter id="PhoneDisplay"
            areaCode="415"
            formatString="###-####"/>
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data-->
    <s:TextInput id="myTI"
        initialize="myTI.text=PhoneDisplay.format(newNumber);"/>
</s:Application>
```

At run time, the following text is displayed:

(415) 123-4567

Error handling: PhoneFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the error property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	**When error occurs**
`Invalid value`	• An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String. • The value contains a different number of digits than what is specified in the format string.
`Invalid format`	• One or more of the characters in the `formatString` do not match the allowed characters specified in the `validPatternChars` property. • The `areaCodeFormat` property is specified but does not contain exactly three numeric placeholders.

Formatting zip codes with an MX formatter

The ZipCodeFormatter class lets you format five-digit or nine-digit United States ZIP codes and six-character Canadian postal codes. The ZipCodeFormatter class’s `formatString` property accepts a formatted string as a definition of the format pattern. The `formatString` property is optional. If it is omitted, the default value of `#####` is used.

The number of digits in the value to be formatted and the value of the `formatString` property must be five or nine for United States ZIP codes, and six for Canadian postal codes.

The following table shows common `formatString` values, input values, and output values:

formatString value	Input	Output	Format
#####	94117, 941171234	94117, 94117	Five-digit U.S. ZIP code
#####-####	941171234, 94117	94117-1234, 94117-0000	Nine-digit U.S. ZIP code
### ###	A1B2C3	A1B 2C3	Six-character Canadian postal code

For United States ZIP codes, if a nine-digit format is requested and a five-digit value is supplied, `-0000` is appended to the value to make it compliant with the nine-digit format. Inversely, if a nine-digit value is supplied for a five-digit format, the number is truncated to five digits.

For Canadian postal codes, only a six-digit value is allowed for either the `formatString` or the input value.

Note: *For United States ZIP codes, only numeric characters are valid. For Canadian postal codes, alphanumeric characters are allowed. Alphabetic characters must be in uppercase.*

Example: Using the ZipCodeFormatter class

The following example uses the ZipCodeFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainZipFormatter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            // Define variable to hold the ZIP code.
            [Bindable]
            private var storedZipCode:Number=123456789;
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- Declare a ZipCodeFormatter and define parameters.-->
        <mx:ZipCodeFormatter id="ZipCodeDisplay"
            formatString="#####-####"/>
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data.-->
    <s:TextInput text="{ZipCodeDisplay.format(storedZipCode)}"/>
</s:Application>
```

At run time, the following text is displayed:

12345-6789

Error handling: ZipCodeFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the error property. An error refers to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	• An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String, except for Canadian postal codes, which allow alphanumeric values. • The number of digits does not match the allowed digits from the `formatString` property.
`Invalid format`	• One or more of the characters in the `formatString` do not match the allowed characters specified in the `validFormatChars` property. • The number of numeric placeholders does not equal 9, 5, or 6.

Deep linking

About deep linking

One of the benefits of an application built with Flex is that it can smoothly transition from state to state without having to fetch a new page from the server and refresh the browser. By avoiding the constant refreshing of pages, the end-user's experience is more fluid and continuous. In addition, the load on the server is greatly reduced because it need only return the application once, rather than a new page every time the user changes views.

However one of the advantages of a browser's page-oriented model is that an application's navigational state is usually clearly coupled to a URL. Thus, when the state of an application changes, the user can usually do the following:

- Bookmark the URL to get back to that state in the application
- Email the URL to a friend
- Use the Back and Forward buttons to navigate to recently visited application states

Losing the ability to couple a browser-managed URL to a specific state of an application can be a fairly big drawback when creating applications. Deep linking adds URL-mapping or “bookmarking” capabilities to applications. At the high level, deep linking lets you do the following:

- Update the URL when the application state changes. This also generates a new entry in the browser’s history.
- Read in values from the URL to change the state of the application. For example, a bookmark could load the application at a particular state.
- Recognize when the URL in the browser’s address bar has changed. If the user clicks the Forward or Back button, or changes the URL in the address bar, the application is notified and can react to the change.

What states of your application are bookmarkable will vary by application, but possibilities include a login page, a product page, a search result page, or a drill down view into data.

Deep linking only works with certain browsers. The following browsers support deep linking:

- Microsoft Internet Explorer version 6 or later
- FireFox for Windows and Macintosh
- Safari for Macintosh

In addition to requiring these browsers, deep linking requires that the client browser also has JavaScript enabled. Deep linking does not work with the standalone Adobe® Flash® Player or Adobe AIR™.

How deep linking works

Deep linking relies on communication between the browser and the application. The communication is bidirectional: if a change occurs in the application, the browser must be notified, and if a change in the browser occurs, then the application must be notified. This communication is handled by the BrowserManager class. This class uses methods in the HTML wrapper's JavaScript to handle events, update the browser's address bar, and call other methods. Another class, URLUtil, is provided to make it easier to parse the URL as you read it in your application and write it back to the browser.

To use deep linking, you write ActionScript that sets and gets portions of the URL to determine which state the application is in. These portions are called fragments, and occur after the pound sign ("#") in the URL. In the following example, `view=1` is the fragment:

```
http://my.domain.com#view=1
```

If the user navigates to a new view in your application, you update the URL and set `view=2` in the URL fragment. If the user then clicks the browser's Back button, the BrowserManager is notified of the event, and gives you a chance to change the application's state in respond to the URL fragment changing back to `view=1`.

You typically use the methods of the URLUtil class to parse the URL. This class provides methods for detecting the server name, port number, and protocol of a URL. In addition, you can use the `objectToString()` method to convert an ActionScript object to a String that you then append to the end of a URL. Alternatively, you can convert any number of name/value pairs on a query string into an object by using the URLUtil class's `stringToObject()` method. This lets you then manipulate the fragment more easily in your application logic.

Deploying applications that use deep linking

To use deep linking, your HTML wrapper requires that the following files be deployed with your application:

- history.css
- history.js
- historyFrame.html

You must include the history.js and history.css files in your wrapper. The following example imports the JS and CSS files:

```
<!--  BEGIN Browser History required section -->
<link rel="stylesheet" type="text/css" href="history/history.css"/>
<script src="history/history.js" language="javascript"></script>
<!--  END Browser History required section -->
```

You must also deploy the historyFrame.html file with your application. This file must be located in the /history sub-directory; its location is relative to your deployed application SWF file's location.

For Adobe®Flash® Builder™, the history.css, history.js, and historyFrame.html files are located in the *flash_builder_install*/sdks/4.6.0/templates/swfobject/history directory.

For the SDK, the files are located in the *sdk_install*/templates/swfobject/history directory.

Flash Builder generates the history.css, history.js, and historyFrame.html files in your project's output directory if you enable deep linking in your project.

Enable deep linking in Flash Builder

1 Select Project > Properties.

2 Select the Flex Compiler option.

3 Select the "Enable integration with browser navigation" option.

Using the BrowserManager

The BrowserManager is a Singleton class that acts as a proxy between the browser and the application. It provides access to the URL in the browser address bar similar to accessing the `document.location` property in JavaScript. When the URL changes in the browser, the BrowserManager is notified of the event. You can then change the URL, respond to the event, or block the event.

To get a reference to the BrowserManager, you call its `getInstance()` method. This method returns the current instance of the manager, which implements IBrowserManager. You can then call methods on the manager such as `setTitle()` and `setFragment()`.

You can also listen to events on that manager. The events are `browserURLChange`, `urlChange`, and `applicationURLChange`. These events are described in "About BrowserManager events" on page 2029.

You can also access properties of the manager. These properties store the browser's current title, plus the full URL and its sub-parts, `fragment` and `base`. The properties are read-only, but you can use methods of the manager, such as `setTitle()` and `setFragment()`, to set the values of some of them.

You typically call the `getInstance()` method when your application initializes. This returns an instance of the Singleton BrowserManager. You then register an event listener for the `browserURLChange` event. This event is dispatched when the user clicks the Back or Forward button in the browser. Finally, you call the `init()` method to initialize the BrowserManager. The first parameter of the `init()` method defines the default fragment, and the second parameter defines the title for the current page.

The following example instantiates the BrowserManager, registers the `parseURL()` method to listen for `browserURLChange` events, and calls the `init()` method with a blank fragment as the default fragment. It sets the value of the page's title to "Test Deep Linking":

```
private function initApp():void {
    browserManager = BrowserManager.getInstance();
    browserManager.addEventListener(BrowserChangeEvent.BROWSER_URL_CHANGE, parseURL);
    browserManager.init("", "Test Deep Linking");
}
```

Calling the BrowserManager class's `init()` method also gets the current URL as a property in the BrowserManager.

If you set a value for the default fragment in the `init()` method, when the URL changes, the URL with the default fragment will be entered in the browser's history. This way, if the user clicks the Back button or in some other way accesses this page from the browser's history, this fragment will be used. Also, the BrowserManager returns the default fragment if there is nothing after the pound sign ("#") in the URL.

Updating the URL

You use the BrowserManager's `setFragment()` method to update the URL in the browser's address bar. You can only change the fragments in the URL. You cannot change the base of the URL, including the server, protocol, or port numbers.

When you use the `setFragment()` method to change the URL, you trigger an `applicationURLChange` event.

The following example updates the URL in the browser whenever you change the active panel in the TabNavigator container. It also keeps a record of the current URL and previous URL each time an `applicationURLChange` event is triggered.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/UpdateURLExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
    <![CDATA[
        import mx.events.BrowserChangeEvent;
        import mx.managers.IBrowserManager;
        import mx.managers.BrowserManager;
        import mx.utils.URLUtil;
        private var browserManager:IBrowserManager;
        private function init():void {
            browserManager = BrowserManager.getInstance();
            browserManager.addEventListener(BrowserChangeEvent.APPLICATION_URL_CHANGE,
                logURLChange);
            browserManager.init("", "Welcome!");
        }
        public function updateTitle(e:Event):void {
            browserManager.setTitle("Welcome " + ti1.text + " from " + ti2.text + "!");
        }
        private function updateURL(event:Event):void {
            var s:String = "panel=" + event.currentTarget.selectedIndex;
            browserManager.setFragment(s);
        }
        private function logURLChange(event:BrowserChangeEvent):void {
            ta1.text += "APPLICATION_URL_CHANGE event:\n";
            ta1.text += " url: " + event.url + "\n"; // Current URL in the browser.
            ta1.text += " prev: " + event.lastURL + "\n"; // Previous URL.
        }
    ]]>
    </fx:Script>
    <mx:TabNavigator id="tn" width="300" change="updateURL(event)">
            <mx:Panel label="Personal Data">
                <mx:Form>
                    <mx:FormItem label="Name:">
                        <mx:TextInput id="ti1"/>
                    </mx:FormItem>
                    <mx:FormItem label="Hometown:">
                        <mx:TextInput id="ti2"/>
                    </mx:FormItem>
                    <mx:Button id="b1" click="updateTitle(event)" label="Submit"/>
                </mx:Form>
            </mx:Panel>
            <mx:Panel label="Credit Card Info">
                <mx:Form>
                    <mx:FormItem label="Type:">
```

```
                        <mx:ComboBox>
                            <mx:dataProvider>
                                <fx:String>Visa</fx:String>
                                <fx:String>MasterCard</fx:String>
                                <fx:String>American Express</fx:String>
                            </mx:dataProvider>
                        </mx:ComboBox>
                    </mx:FormItem>
                    <mx:FormItem label="Number:">
                        <mx:TextInput id="ccnumber"/>
                    </mx:FormItem>
                </mx:Form>
            </mx:Panel>
            <mx:Panel label="Check Out">
              <mx:TextArea id="ta2" text="You must agree to all the following conditions..."/>
                <mx:CheckBox label="Agree"/>
            </mx:Panel>
    </mx:TabNavigator>
    <mx:TextArea id="ta1" width="580" height="400"/>
</s:Application>
```

When you click on the second panel, you trigger a `change` event, which causes Flash Player to call the `updateURL()` method. This method calls the `setFragment()` method that changes the URL in the browser's address bar. If you cycle through the panels, the URL in the browser's address bar will change from this:

```
http://localhost:8100/devapps/code/deeplinking/FragmentExample.html#panel=1
```

To this:

```
http://localhost:8100/devapps/code/deeplinking/FragmentExample.html#panel=2
```

And then, to this:

```
http://localhost:8100/devapps/code/deeplinking/FragmentExample.html#panel=0
```

Each time the application changes the URL in the browser's address bar, Flash Player dispatches an `applicationURLChange` event. This example logs the previous and current URLs, which are properties of this event object.

Parsing the URL

Changing the URL in the browser's address bar does not necessarily provide you with deep linking functionality. For example, loading a bookmarked URL that specifies the panel would not start the application on that panel. You must add code to the application that parses the URL so that the application's state reflects the URL.

In this case, you add a listener for the `browserURLChange` event. In that listener, you parse the URL and set the application state according to the results. For example, if the URL includes a fragment such as `panel=2`, then you write code that sets the current panel with the selected index of 2.

The `browserURLChange` event is not triggered when the application first loads. As a result, you must also check the URL on startup with the application's `creationComplete` event, and trigger the parsing before the application finishes being rendered on the screen.

The following is a simple example that sets the value of the Accordion's `selectedIndex` property to the value of the `index` fragment in the URL. You request this application by setting the value of the `index` fragment on the URL, as the following example shows:

```
http://www.myurl.com/InitFrag.html#index=1
```

When the application starts up, the Accordion is opened to the panel corresponding to the value of the `index` fragment:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/InitFrag.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init(event);">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.managers.BrowserManager;
            import mx.managers.IBrowserManager;
            import mx.events.BrowserChangeEvent;
            import mx.utils.URLUtil;
            private var bm:IBrowserManager;
            private function init(e:Event):void {
                bm = BrowserManager.getInstance();
                bm.init("", "Welcome!");
                parseURL(e);
            }

            [Bindable]
            private var indexFromURL:int;
            private function parseURL(e:Event):void {
                var o:Object = URLUtil.stringToObject(bm.fragment);
                indexFromURL = o.index;
            }
        ]]>
    </fx:Script>
    <mx:Accordion selectedIndex="{indexFromURL}">
        <mx:VBox label="Panel 1">
            <mx:Label text="Accordion container panel 1"/>
        </mx:VBox>
        <mx:VBox label="Panel 2">
            <mx:Label text="Accordion container panel 2"/>
        </mx:VBox>
        <mx:VBox label="Panel 3">
            <mx:Label text="Accordion container panel 3"/>
        </mx:VBox>
    </mx:Accordion>
</s:Application>
```

The following example expands on the example from “Updating the URL” on page 2024. It reads the URL on startup, and opens the application to the first, second, or third panel, depending on the value of the `panel` fragment. It also sets the title of the HTML page according to which panel is opened.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/TabNavExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp()"
    height="250"
    width="500">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
    <![CDATA[
        import mx.events.BrowserChangeEvent;
        import mx.managers.IBrowserManager;
        import mx.managers.BrowserManager;
        import mx.utils.URLUtil;
        public var browserManager:IBrowserManager;
        private function initApp():void {
            browserManager = BrowserManager.getInstance();
            browserManager.addEventListener(BrowserChangeEvent.BROWSER_URL_CHANGE, parseURL);
            browserManager.init("", "Shipping");
        }
        private var parsing:Boolean = false;
        private function parseURL(event:Event):void {
            parsing = true;
            var o:Object = URLUtil.stringToObject(browserManager.fragment);
            if (o.view == undefined)
                o.view = 0;
            tn.selectedIndex = o.view;
            browserManager.setTitle((tn.selectedIndex == 0) ? "Shipping" : "Receiving");
            tn.validateNow();
            var details:Boolean = o.details == true;
            if (tn.selectedIndex == 0)
                shipDetails.selected = details;
            else
                recvDetails.selected = details;
            parsing = false;
        }
        private function updateURL():void {
            if (!parsing)
                callLater(actuallyUpdateURL);
        }
        private function actuallyUpdateURL():void {
            var o:Object = {};
            var t:String = "";
            if (tn.selectedIndex == 1) {
                t = "Receiving";
                o.view = tn.selectedIndex;
                if (recvDetails.selected)
                    o.details = true;
            } else {
                t = "Shipping";
                o.view = tn.selectedIndex;
                if (shipDetails.selected)
                    o.details = true;
            }
            var s:String = URLUtil.objectToString(o);
            browserManager.setFragment(s);
            browserManager.setTitle(t);
        }
    ]]>
    </fx:Script>
    <mx:TabNavigator id="tn" change="updateURL()" width="300">
            <mx:Panel label="Shipping">
                <mx:CheckBox id="shipDetails" label="Show Details" change="updateURL()" />
            </mx:Panel>
            <mx:Panel label="Receiving">
                <mx:CheckBox id="recvDetails" label="Show Details" change="updateURL()" />
            </mx:Panel>
    </mx:TabNavigator>
</s:Application>
```

This example has one major drawback: it does not “remember” the state of the panel that is not in the current view. If you copy the bookmark and open it in a new browser, the view that you were last looking at, and the state of the CheckBox on that view, are maintained. However, the CheckBox in the other view that was hidden is reset to its original value (unchecked). There are several techniques to solve this issue. For more information, see “Using deep linking with navigator containers” on page 2033.

About BrowserManager events

The BrowserManager triggers the following types of BrowserChangeEvent events:

- `applicationURLChange`
- `browserURLChange`
- `urlChange`

These changes can be triggered by the following actions:

- URL is changed programmatically, such as with the BrowserManager’s `setFragment()` method (`applicationURLChange`)
- User clicks the Forward or Back button (`browserURLChange`)
- User changes the URL and clicks Enter or Go (`browserURLChange`)

The `urlChange` event is triggered whenever either `applicationURLChange` or `browserURLChange` events are triggered.

The following example shows the properties of the change events in a DataGrid control. You can trigger an `applicationURLChange` event by selecting a new value in the ComboBox control. You can trigger a `browserURLChange` event by using the Forward and Back buttons in your browser. In both cases, you also trigger a `urlChange` event.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/URLChangeLogger.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
    <![CDATA[
        import mx.managers.BrowserManager;
        import mx.managers.IBrowserManager;
        import mx.events.BrowserChangeEvent;
        private var bm:IBrowserManager;
        private function init():void {
            bm = BrowserManager.getInstance();
            bm.addEventListener(BrowserChangeEvent.APPLICATION_URL_CHANGE, doEvent);
            bm.addEventListener(BrowserChangeEvent.BROWSER_URL_CHANGE, doEvent);
            bm.addEventListener(BrowserChangeEvent.URL_CHANGE, doEvent);
            bm.init("", "Base title");
        }
        public function doEvent(evt:BrowserChangeEvent):void {
            eventDG.dataProvider.addItem(evt);
        }
    ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:Array id="dp">
            <fx:Object label="one"/>
            <fx:Object label="two"/>
            <fx:Object label="three"/>
        </fx:Array>
    </fx:Declarations>
    <mx:ComboBox id="cb" dataProvider="{dp}"
        change="bm.setFragment('selectedItem=' + cb.selectedItem.label);"/>

    <mx:DataGrid id="eventDG"
        dataProvider="[]"
        width="100%"
        variableRowHeight="true"
        wordWrap="true"
        height="500"/>
</s:Application>
```

Changing the URL fragments in the browser's address bar triggers a `browserURLChange` event but does not trigger a page reload in FireFox or Internet Explorer 7. In Internet Explorer version 6 and earlier, changing the part of the address that is to the right of the pound sign ("#") *does* trigger a page reload.

Setting the title of the HTML wrapper

You can use the BrowserManager's `setTitle()` method to set the title in your HTML wrapper. This shows up as the name of the web page in the title bar of the browser. When you first initialize the BrowserManager, you set the value of the title in the second parameter to the `init()` method.

The following example sets the initial value of the title to "Welcome". It then changes the title depending on the name and hometown that you enter in the TextInput fields.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/TitleManipulationExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
    <![CDATA[
        import mx.managers.BrowserManager;
        import mx.managers.IBrowserManager;
        import mx.events.BrowserChangeEvent;
        private var bm:IBrowserManager;
        private function init():void {
            bm = BrowserManager.getInstance();
            bm.init("", "Welcome!");
        }
        public function updateTitle(e:Event):void {
            bm.setTitle("Welcome " + ti1.text + " from " + ti2.text + "!");
        }
    ]]>
    </fx:Script>

    <mx:Form>
        <mx:FormItem label="Name:">
            <mx:TextInput id="ti1"/>
        </mx:FormItem>
        <mx:FormItem label="Hometown:">
            <mx:TextInput id="ti2"/>
        </mx:FormItem>
        <mx:Button id="b1" click="updateTitle(event)" label="Submit"/>
    </mx:Form>
</s:Application>
```

Passing request data with URL fragments

In applications, there are several ways to pass values to an application with the URL. You can convert query string parameters to `flashVars` variables or you can append them to the SWF file's URL (for example, MyApp.swf?value1=x&value2=y). You do both of these things in the HTML wrapper that embeds the application. To access the values in your application, you then use the `FlexGlobals.topLevelApplication.application` object. For more information on passing `flashVars` variables, see "Passing request data with flashVars properties" on page 229.

The BrowserManager also provides a method of accessing values from a URL inside your application. You do this by accessing the URL fragments that deep linking uses. By default, you can append any Strings after the pound sign ("#") in the URL and be able to access them as semi-colon-separated name/value pairs. For example, with a URL like the following, you can access the `firstName` and `lastName` values once you convert the fragment to an object:

```
http://www.mydomain.com/MyApp.html#firstName=Nick;lastName=Danger
```

To convert these parameters to an object, you get the URL and then use the URLUtil class's `stringToObject()` method. You then access `firstName` and `lastName` as properties on that new object.

To make the URL more "URL-like", you can separate the fragments with an ampersand ("&"). When you convert the fragment to an object with the `stringToObject()` method, you specify the new delimiter.

The following example takes a URL that sets the value of the `firstName` and `lastName` query string parameters in its fragment. It specifies an ampersand as the delimiter.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/PassURLParamsAsFragments.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init(event);">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
    <![CDATA[
        import mx.managers.BrowserManager;
        import mx.managers.IBrowserManager;
        import mx.utils.URLUtil;
        private var bm:IBrowserManager;
        [Bindable]
        private var fName:String;
        [Bindable]
        private var lName:String;
        private function init(e:Event):void {
            bm = BrowserManager.getInstance();
            bm.init("", "Welcome!");
            /* The following code will parse a URL that passes firstName and lastName as
               query string parameters after the "#" sign; for example:
               http://www.mydomain.com/MyApp.html#firstName=Nick&lastName=Danger */
            var o:Object = URLUtil.stringToObject(bm.fragment, "&");
            fName = o.firstName;
            lName = o.lastName;
        }
    ]]>
    </fx:Script>

    <mx:Form>
        <mx:FormItem label="First name:">
            <mx:Label id="ti1" text="{fName}"/>
        </mx:FormItem>
        <mx:FormItem label="Last name:">
            <mx:Label id="ti2" text="{lName}"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

Using deep linking with navigator containers

A common use case is for a user to start filling out a form, and then either bookmark the form for later use, or send the URL to someone else to take a look at the values. In many cases, forms are implemented as navigator containers such as TabNavigator and Accordion containers.

This can be a problem for deep linking because you might try to read properties on controls that are in views that have not yet been viewed (and therefore, the controls have not yet been instantiated). One possible solution is to disable deferred instantiation, and set the value of the container's `creationPolicy` property to `all`. This instructs Flex to instantiate all controls in all views at application startup, regardless of whether they can be viewed initially or not. This is not a recommended solution because it creates additional overhead when the application starts, and can end up using unnecessary amounts of processor time and memory. Depending on the complexity of the user interface, setting the value of the `creationPolicy` property to `all` can seriously degrade performance of your application and detract from a positive user experience. For more information about creation policies, see "About the creationPolicy property" on page 2336.

A technique that does not rely on deferred instantiation is to bind the values of the properties you want to maintain state on to the values in the URL. Controls that have not yet been created apply these values when they are created. This has the benefit of keeping the URL and the application's state in sync with each other, without requiring that a view's controls are first created.

You do this by setting up variables that are mapped to the URL's values at start up and then kept in sync with the application as the user interacts with it. When the application starts up, you set the value of the bound property to the value taken from the URL (if there is one). Then, whenever the application's state changes (such as when the user navigates to a new panel or clicks on a check box), you update the bound property to the new value. You also update the value of the property's fragment in the URL.

By doing this, though, you must wrap your variable assignments with try/catch blocks because you will attempt to set the values of variables with properties of components that might not yet have been instantiated. This ensures that assignment errors are caught rather than thrown as run-time errors.

The following example uses a TabNavigator container to help the user through a payment process. Whenever the user navigates to a new panel, changes the value in a TextInput control, or checks the CheckBox control, the URL and the bound properties are updated. At any time, you can bookmark the URL, close your browser, and then come back to the application at a later time. The values of all the properties are maintained in the bookmarked URL.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/ComplexMultiPanelExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="init();parseURL(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
    <![CDATA[
        import mx.events.BrowserChangeEvent;
        import mx.managers.IBrowserManager;
        import mx.managers.BrowserManager;
        import mx.utils.URLUtil;
        private var bm:IBrowserManager;
        [Bindable]
        private var agreeBoxFromURL:Boolean;
```

```
[Bindable]
private var personNameFromURL:String;
[Bindable]
private var hometownFromURL:String;
[Bindable]
private var cctypeFromURL:int;
[Bindable]
private var ccnumberFromURL:String;
private function init():void {
    bm = BrowserManager.getInstance();
  bm.addEventListener(BrowserChangeEvent.BROWSER_URL_CHANGE, parseURL);
    bm.init("", "Welcome!");
}
/* This method is called once when application starts up. It is also
   called when the browser's address bar changes, either due to user action
   or user navigation with the browser's Forward and Back buttons. */
private function parseURL(event:Event):void {
    var o:Object = URLUtil.stringToObject(bm.fragment, "&");
    if (o.panel == undefined)
        o.panel = 0;
    tn.selectedIndex = o.panel;
    tn.validateNow();
    personNameFromURL = o.personName;
    hometownFromURL = o.hometown;
    ccnumberFromURL = o.ccnumber;
    cctypeFromURL = o.cctype;
    agreeBoxFromURL = o.agreeBox;
}
public function updateTitle(e:Event):void {
    l1.text += "updateTitle()\n";
    bm.setTitle("Welcome " + personName.text + " from " + hometown.text + "!");
}
private function updateURL():void {
    /* Called when state changes in the application, such as when the panel changes,
       or a checkbox is checked.
       You must wrap the following assignments in a try/catch block, otherwise the
       application tries to access components that have not yet been created.
       You can circumvent this by setting the container's creationPolicy to "all",
       but that is not a good solution for performance reasons. */
    try {
        personNameFromURL = personName.text;
        hometownFromURL = hometown.text;
        ccnumberFromURL = ccnumber.text;
        cctypeFromURL = cctype.selectedIndex;
        agreeBoxFromURL = agreeBox.selected;
    } catch (e:Error) {
    }
    var o:Object = {};
    try {
        o.panel = tn.selectedIndex;
        o.personName = personName.text;
        o.hometown = hometown.text;
        o.ccnumber = ccnumber.text;
        o.cctype = cctype.selectedIndex;
        o.agreeBox = agreeBox.selected;
    } catch (e:Error) {
    } finally {
```

```
                var s:String = URLUtil.objectToString(o, "&");
                bm.setFragment(s);
            }
        }
    ]]>
    </fx:Script>
    <mx:TabNavigator id="tn" width="300" change="updateURL()">
        <mx:Panel label="Personal Data">
            <mx:Form>
                <mx:FormItem label="Name:">
                    <mx:TextInput id="personName"
                        text="{personNameFromURL}"
                        focusOut="updateURL()"
                        enter="updateURL()"/>
                </mx:FormItem>
                <mx:FormItem label="Hometown:">
                    <mx:TextInput id="hometown"
                        text="{hometownFromURL}"
                        focusOut="updateURL()"
                        enter="updateURL()"/>
                </mx:FormItem>
                <mx:Button id="b1" click="updateTitle(event)" label="Submit"/>
            </mx:Form>
        </mx:Panel>
        <mx:Panel label="Credit Card Info">
            <mx:Form>
                <mx:FormItem label="Type:">
                    <mx:ComboBox id="cctype"
                        change="updateURL()"
                        selectedIndex="{cctypeFromURL}">
                        <mx:dataProvider>
                            <fx:String>Visa</fx:String>
                            <fx:String>MasterCard</fx:String>
                            <fx:String>American Express</fx:String>
                        </mx:dataProvider>
                    </mx:ComboBox>
                </mx:FormItem>
                <mx:FormItem label="Number:">
                    <mx:TextInput id="ccnumber"
                        text="{ccnumberFromURL}"
                        focusOut="updateURL()"
                        enter="updateURL()"/>
                </mx:FormItem>
            </mx:Form>
        </mx:Panel>
        <mx:Panel label="Check Out">
            <mx:TextArea id="ta2" text="You must agree to all the following conditions..."/>
            <mx:CheckBox id="agreeBox"
                label="Agree"
                selected="{agreeBoxFromURL}"
                click="updateURL()"/>
        </mx:Panel>
    </mx:TabNavigator>
    <mx:TextArea id="l1" height="400" width="300"/>
</s:Application>
```

Rather than update the values of the bindable variables in the `updateURL()` method, you can also set their values in the event handlers. For example, when the user changes the value of the TextInput control, you can use the `focusOut` and `enter` event handlers to set the value of the `hometownFromURL` property:

```
<mx:TextInput id="hometown"
    text="{hometownFromURL}"
    focusOut="hometownFromURL=hometown.text;updateURL()"
    enter="hometownFromURL=hometown.text;updateURL()"/>
```

In this example, the controls in the view update the model when the controls' properties change. This helps enforce a cleaner separation between the model and the view.

Accessing information about the current URL

In some cases, it is important to get information about the current URL, such as the server name or the protocol that was used to return the SWF's wrapper. You do this by using the BrowserManager and the URLUtil class.

You use the BrowserManager to get the URL, using either the `url`, `fragment`, or `base` properties. You can then use convenience methods of the URLUtil class to parse the URL. You can use these methods to extract the port, protocol, and server name from the URL. You can also use the URLUtil class to check if the protocol is secure or not.

The following example uses the URLUtil and BrowserManager classes to get information about the URL that was used to return the application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deeplinking/UseURLUtil.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp()"
    height="250" width="500">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
    <![CDATA[
        import mx.utils.URLUtil;
        import mx.managers.IBrowserManager;
        import mx.managers.BrowserManager;
        import mx.events.BrowserChangeEvent;
        public var browserManager:IBrowserManager;
        private function initApp():void {
            browserManager = BrowserManager.getInstance();
       browserManager.addEventListener(BrowserChangeEvent.URL_CHANGE, showURLDetails);
            browserManager.init("", "Welcome!");
        }
        [Bindable]
        private var fullURL:String;
        [Bindable]
        private var baseURL:String;
        [Bindable]
        private var fragment:String;
        [Bindable]
        private var protocol:String;
        [Bindable]
        private var port:int;
        [Bindable]
        private var serverName:String;
        [Bindable]
        private var isSecure:Boolean;
        [Bindable]
        private var previousURL:String;
        private function showURLDetails(e:BrowserChangeEvent):void {
            var url:String = browserManager.url;
            baseURL = browserManager.base;
            fragment = browserManager.fragment;
            previousURL = e.lastURL;
            fullURL = mx.utils.URLUtil.getFullURL(url, url);
            port = mx.utils.URLUtil.getPort(url);
            protocol = mx.utils.URLUtil.getProtocol(url);
            serverName = mx.utils.URLUtil.getServerName(url);
            isSecure = mx.utils.URLUtil.isHttpsURL(url);
        }
    ]]>
    </fx:Script>
    <mx:Form>
        <mx:FormItem label="Full URL:">
            <mx:Label text="{fullURL}"/>
        </mx:FormItem>
        <mx:FormItem label="Base URL:">
            <mx:Label text="{baseURL}"/>
        </mx:FormItem>
        <mx:FormItem label="Fragment:">
            <mx:Label text="{fragment}"/>
        </mx:FormItem>
        <mx:FormItem label="Protocol:">
            <mx:Label text="{protocol}"/>
        </mx:FormItem>
        <mx:FormItem label="Port:">
            <mx:Label text="{port}"/>
        </mx:FormItem>
        <mx:FormItem label="Server name:">
            <mx:Label text="{serverName}"/>
        </mx:FormItem>
        <mx:FormItem label="Is secure?:">
            <mx:Label text="{isSecure}"/>
        </mx:FormItem>
        <mx:FormItem label="Previous URL:">
            <mx:Label text="{previousURL}"/>
        </mx:FormItem>
    </mx:Form>
</s:Application>
```

Printing

Many applications built in Adobe® Flex® let users print from within the application. For example, you might have an application that returns confirmation information after a user completes a purchase. Your application can allow users to print the information on the page to keep for their records.

***Note:** You can also print by using the context menu in Adobe® Flash® Player, or the Flash ActionScript PrintJob class, which is documented in the ActionScript 3.0 Reference for the Adobe Flash Platform.*

About printing by using Flex classes

The Flex mx.printing package contains classes that facilitate the creation of printing output from Flex applications:

FlexPrintJob A class that prints one or more objects. Automatically splits large objects for printing on multiple pages and scales the output to fit the page size.

PrintDataGrid A subclass of the DataGrid control with a default appearance that is customized for printing. The class includes properties and a method that provide additional sizing and printing features.

PrintAdvancedDataGrid A subclass of the AdvancedDataGrid control with a default appearance that is customized for printing. The class includes properties and a method that provide additional sizing and printing features.

PrintOLAPDataGrid A subclass of the OLAPDataGrid control with a default appearance that is customized for printing. The class includes properties and a method that provide additional sizing and printing features.

FlexPrintJobScaleType Defines constants used in the FlexPrintJob `addObject()` method.

Together, these classes give you control over how the user prints information from the application. For example, your application can print only a selected subset of the information on the screen, or it can print information that is not being displayed. Also, your application can reformat the information and optimize its layout and appearance for printing.

Users can print to PostScript and non-PostScript printers, including the Adobe® PDF® and Adobe® FlashPaper™ from Adobe® printer drivers.

Using the FlexPrintJob class

You use the FlexPrintJob class to print one or more Flex objects, such as a Form or VBox container. For each object that you specify, Flex prints the object and all objects that it contains. The objects can be all or part of the displayed interface, or they can be components that format data specifically for printing. The FlexPrintJob class lets you scale the output to fit the page, and automatically uses multiple pages to print an object that does not fit on a single page.

You use the FlexPrintJob class to print a dynamically rendered document that you format specifically for printing. This capability is especially useful for rendering and printing such information as receipts, itineraries, and other displays that contain external dynamic content, such as database content and dynamic text.

You often use the FlexPrintJob class within an event listener. For example, you can use a Button control with an event listener that prints some or all of the application.

***Note:** The FlexPrintJob class causes the operating system to display a Print dialog box. You cannot print without some user action.*

Build and send a print job

You print output by building and sending a print job.

1 Create an instance of the FlexPrintJob class:

```
var printJob:FlexPrintJob = new FlexPrintJob();
```

2 Start the print job:

```
printJob.start();
```

This causes the operating system to display a Print dialog box.

3 Add one or more objects to the print job and specify how to scale them:

```
printJob.addObject(myObject, FlexPrintJobScaleType.MATCH_WIDTH);
```

Each object starts on a new page.

4 Send the print job to the printer:

```
printJob.send();
```

5 Free up any unneeded objects.

***Note:** Because you are spooling a print job to the user's operating system between your calls to the `start()` and `send()` methods, you should limit the code between these calls to print-specific activities. For example, the content should not interact with the user between the `start()` and `send()` methods.*

The following sections detail the procedures to use in these steps.

Starting a print job

To start a print job, you create an instance of the FlexPrintJob class and call its `start()` method. This method prompts Flash Player or Adobe® AIR™ to spool the print job to the user's operating system, which causes the user's operating system to display a Print dialog box.

If the user selects an option to begin printing from the Print dialog box, the `start`() method returns a value of `true`. If the user cancels the print job, the return value is `false`. After the user exits the operating system Print dialog box, the `start()` method uses the printer information to set values for the FlexPrintJob object's `pageHeight` and `pageWidth` properties, which represent the dimensions of the printed page area.

***Note:** Depending on the user's operating system, an additional dialog box might appear until spooling is complete and the application calls the `send()` method.*

Only one print job can be active at a time. You cannot start a second print job until one of the following has happened with the previous print job:

- The `start()` method returns a value of `false` (the job failed).
- The `send()` method completes execution following a successful call to the `addObject()` method. Because the `send()` method is synchronous, code that follows it can assume that the call completed successfully.

Adding objects to the print job

You use the `addObject()` method of the FlexPrintJob class to add objects to the print job. Each object starts on a new page; therefore, the following code prints a DataGrid control and a Button control on separate pages:

```
printJob.addObject(myDataGrid);
printJob.addObject(myButton);
```

Scaling a print job

The *scaleType* parameter of the `addObject()` method determines how to scale the output. Use the following FlexPrintJobScaleType class constants to specify the scaling method:

Constant	Action
MATCH_WIDTH	(Default) Scales the object to fill the available page width. If the resulting object height exceeds the page height, the output spans multiple pages.
MATCH_HEIGHT	Scales the object to fill the available page height. If the resulting object width exceeds the page width, the output spans multiple pages.
SHOW_ALL	Scales the object to fit on a single page, filling one dimension; that is, it selects the smaller of the MATCH_WIDTH or MATCH_HEIGHT scale types.
FILL_PAGE	Scales the object to fill at least one page completely; that is, it selects the larger of the MATCH_WIDTH or MATCH_HEIGHT scale types.
NONE	Does not scale the output. The printed page has the same dimensions as the object on the screen. If the object height, width, or both dimensions exceed the page width or height, the output spans multiple pages.

If an object requires multiple pages, the output splits at the page boundaries. This can result in unreadable text or inappropriately split graphics. For information on how to format your print job to avoid these problems, see “Printing multipage output” on page 2045.

The FlexPrintJob class includes two properties that can help your application determine how to scale the print job. These properties are read-only and are initially 0. When the application calls the `start()` method and the user selects the Print option in the operating system Print dialog box, Flash Player or AIR retrieves the print settings from the operating system. The `start()` method populates the following properties:

Property	Type	Unit	Description
`pageHeight`	Number	Points	Height of the printable area on the page; does not include any user-set margins.
`pageWidth`	Number	Points	Width of the printable area on the page; does not include any user-set margins.

***Note:** A point is a print unit of measurement that is 1/72 of an inch. Flex automatically maps 72 pixels to one inch (72 points) of printed output, based on the printer settings.*

Completing the print operation

To send the print job to a printer after using the FlexPrintJob `addObject()` method, use the `send()` method, which causes Flash Player or AIR to stop spooling the print job so that the printer starts printing.

After sending the print job to a printer, if you use print-only components to format output for printing, call `removeChild()` to remove the print-specific component. For more information, see “Using a print-specific output format” on page 2042.

Example: A simple print job

The following example prints a DataGrid object exactly as it appears on the screen, without scaling:

```
<?xml version="1.0"?>
<!-- printing\DGPrint.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.printing.*;
            // Create a PrintJob instance.
            private function doPrint():void {
                // Create an instance of the FlexPrintJob class.
                var printJob:FlexPrintJob = new FlexPrintJob();
                // Start the print job.
                if (printJob.start() != true) return;
                // Add the object to print. Do not scale it.
                printJob.addObject(myDataGrid, FlexPrintJobScaleType.NONE);
                // Send the job to the printer.
                printJob.send();
            }
        ]]>
    </fx:Script>
    <mx:VBox id="myVBox">
        <mx:DataGrid id="myDataGrid" width="300">
            <mx:dataProvider>
                <fx:Object Product="Flash" Code="1000"/>
                <fx:Object Product="Flex" Code="2000"/>
                <fx:Object Product="ColdFusion" Code="3000"/>
                <fx:Object Product="JRun" Code="4000"/>
            </mx:dataProvider>
        </mx:DataGrid>
        <mx:Button id="myButton"
            label="Print"
            click="doPrint();"/>
    </mx:VBox>
</s:Application>
```

In this example, selecting the Button control invokes the `doPrint()` event listener. The event listener creates an instance of the FlexPrintJob class to print the DataGrid control, adds the DataGrid control to the print job using the `addObject()` method, and then uses the `send()` method to print the page.

To print the DataGrid and Button controls on your page, specify myVBox, the ID of the object that contains both controls, in the `addObject()` method's *object* parameter. If want to print the DataGrid and Button controls on separate pages, specify each object in a separate `addObject()` method.

To print the DataGrid control so that it spans the page width, omit the second `addObject()` method parameter, or specify `FlexPrintJobScaleType.MATCH_WIDTH`. To print the DataGrid control with the largest size that fits on a single page, specify `FlexPrintJobScaleType.SHOW_ALL`. In the previous example, `FlexPrintJobScaleType.SHOW_ALL` has the same result as `FlexPrintJobScaleType.MATCH_WIDTH` because the DataGrid is short.

Print containers with a transparent background

Many Flex containers, such as the Spark Group and MX Box containers, are transparent by default. If you print a transparent container, the background color of anything behind the transparent container prints.

As an alternative to the Spark Group container, use a Spark BorderContainer or a SkinnableContainer with the `backgroundColor` property set. You can also define another container behind the Group with the appropriate background color. Or, set the `backgroundColor` property of the Application container.

For the MX Box and other containers, make sure to set the `backgroundColor` property.

Using a print-specific output format

In most cases, you do not want your printed output to look like the screen display. The screen might use a horizontally oriented layout that is not appropriate for paper printing. You might have display elements on the screen that would be distracting in a printout. You might also have other reasons for wanting the printed and displayed content to differ; for example, you might want to omit a password field.

To print your output with print-specific contents and appearance, use separate objects for the screen layout and the print layout, and share the data used in the screen layout with the print layout. You then use the print layout in your call or calls to the FlexPrintJob `addObject()` method.

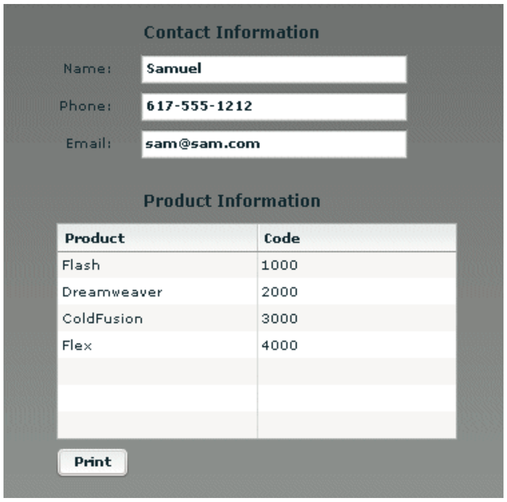
If your form includes a DataGrid, AdvancedDataGrid, or OLAPDataGrid control, use the PrintDataGrid, PrintAdvancedDataGrid, PrintOLAPDataGrid or control in your print layout. The print version of these controls have two advantages over the normal controls for printed output:

- It has a default appearance that is designed specifically for printing.
- It has properties and methods that support printing grids that contain multiple pages of data.

The code in the next section uses a short PrintDataGrid control. For more information on using the PrintDataGrid control, see “Using the PrintDataGrid control for multipage grids” on page 2045. For more information on using the PrintAdvancedDataGrid and PrintOLAPDataGrid control, see “Using the PrintAdvancedDataGrid control” on page 2053.

Example: A simple print-specific output format

This example creates a Flex form with three text boxes for entering contact information, and a data grid that displays several lines of product information. The following image shows how the output of this application looks on the screen:



Contact Information

Name: Samuel

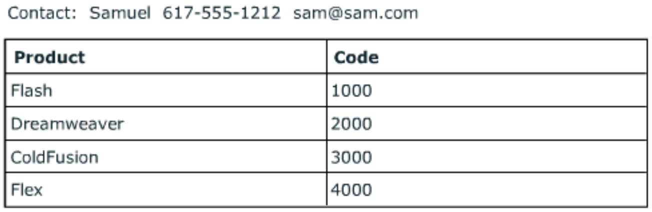
Phone: 617-555-1212

Email: sam@sam.com

Product Information

Product	Code
Flash	1000
Dreamweaver	2000
ColdFusion	3000
Flex	4000

The following image shows how the output looks when the user clicks the Print button to print the data:



Contact: Samuel 617-555-1212 sam@sam.com

Product	Code
Flash	1000
Dreamweaver	2000
ColdFusion	3000
Flex	4000

In this example, the MXML application file displays the screen and controls the printing. A separate custom MXML component defines the appearance of the printed output.

When the user clicks the Print button, the application's `doPrint()` method does the following things:

1 Creates and starts the print job to display the operating system's Print dialog box.

2 After the user starts the print operation in the Print dialog box, creates a child control using the myPrintView component.

3 Sets the MyPrintView control's data from the form data.

4 Sends the print job to the printer.

5 Cleans up memory by removing the print-specific child component.

The printed output does not include the labels from the screen, and the application combines the text from the screen's three input boxes into a single string for printing in a Label control.

The following code shows the contents of the application file:

```
<?xml version="1.0"?>
<!-- printing\DGPrintCustomComp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="450" width="600">
    <fx:Script>
        <![CDATA[
            import mx.printing.FlexPrintJob;
            import myComponents.MyPrintView;
            public function doPrint():void {
                // Create a FlexPrintJob instance.
                var printJob:FlexPrintJob = new FlexPrintJob();

                // Start the print job.
                if(printJob.start()) {
                    // Create a MyPrintView control as a child
                    // of the current view.
                    var formPrintView:MyPrintView = new MyPrintView();
                    addElement(formPrintView);

                    // Populate the print control's contact label
                    // with the text from the form's name,
                    // phone, and e-mail controls.
                    formPrintView.contact.text =
                        "Contact: " + custName.text + "  " +
                        custPhone.text + "  " + custEmail.text;
                    // Set the print control's data grid data provider to be
                    // the displayed data grid's data provider.
                    formPrintView.myDataGrid.dataProvider =
                        myDataGrid.dataProvider;

                    // Add the SimplePrintview control to the print job.
                    // For comparison, try setting the
                    // second parameter to "none".
                    printJob.addObject(formPrintView);

                    // Send the job to the printer.
                    printJob.send();

                    // Remove the print-specific control to free memory.
                    removeElement(formPrintView);
                }
            }
        ]]>
    </fx:Script>
    <!-- The form to display-->
    <mx:Form id="myForm">
        <mx:FormHeading label="Contact Information"/>
        <mx:FormItem label="Name: ">
            <mx:TextInput id="custName"
                width="200"
                text="Samuel Smith"
                fontWeight="bold"/>
        </mx:FormItem>
        <mx:FormItem label="Phone: ">
            <mx:TextInput id="custPhone"
                width="200"
                text="617-555-1212"
                fontWeight="bold"/>
        </mx:FormItem>
        <mx:FormItem label="Email: ">
            <mx:TextInput id="custEmail"
                width="200"
                text="sam@sam.com"
                fontWeight="bold"/>
        </mx:FormItem>
        <mx:FormHeading label="Product Information"/>
        <mx:DataGrid id="myDataGrid" width="300">
            <mx:dataProvider>
                <fx:Object Product="Flash" Code="1000"/>
                <fx:Object Product="Flex" Code="2000"/>
                <fx:Object Product="ColdFusion" Code="3000"/>
                <fx:Object Product="JRun" Code="4000"/>
            </mx:dataProvider>
        </mx:DataGrid>
        <mx:Button id="myButton"
            label="Print"
            click="doPrint();"/>
    </mx:Form>
</s:Application>
```

The following MyPrintView.mxml file defines the component used by the application's `doPrint()` method. The component is a VBox container; it contains a Label control, into which the application writes the contact information from the first three fields of the form, and a PrintDataGrid control, which displays the data from the data source of the screen view's DataGrid control. For more information on the PrintDataGrid control and its advantages for printing, see "Using the PrintDataGrid control for multipage grids" on page 2045.

```
<?xml version="1.0"?>
<!-- printing\myComponents\MyPrintView.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    backgroundColor="#FFFFFF"
    height="300" width="500"
    paddingTop="50" paddingLeft="50" paddingRight="50">
    <!-- The controls to print, a label and a PrintDataGrid control. -->
    <mx:Label id="contact"/>
    <mx:PrintDataGrid id="myDataGrid" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="Product"/>
            <mx:DataGridColumn dataField="Code"/>
        </mx:columns>
    </mx:PrintDataGrid>
</mx:VBox>
```

Printing multipage output

You can print well-formatted multipage output under the following conditions:

- When each control fits on a print page or less. You often encounter such jobs when printing a form with fixed-length fields.
- When the printed output includes one or more PrintDataGrid controls that are too long to print on a single page, particularly if the control height might vary, depending on the data. A good example of this type of output is a customer order receipt, which starts with the customer information, has an indeterminate number of order line items, and ends with total information.

Printing known-length multipage output

If you know the length of each component in a multipage document, you can create a separate print layout component for each page you print, and specify each layout page in a separate `addObject()` method, as follows:

```
printJob.addObject(introPrintView, "ShowAll");
printJob.addObject(finDetailPrintView, "ShowAll");
printJob.addObject(hrDetailPrintView, "ShowAll");
printJob.addObject(summaryPrintView, "ShowAll");
```

Using the PrintDataGrid control for multipage grids

When a DataGrid control with many rows does not fit on a single screen in your application, you typically have scroll bars that let users view all the data. When you print the DataGrid control, the output is the same as the screen display. Therefore, if your DataGrid control has rows or columns that are not immediately visible, they do not print. If you replace the DataGrid control with a PrintDataGrid control that does not have a height specified (or has a large height), you print all the rows, but some rows could be partially printed on the bottom of one page and partially printed at the top of another, as you often see with HTML printout.

You can solve these problems by using the following features of the PrintDataGrid control. These features let you correctly print grids that contain multiple pages of data without splitting rows across pages:

sizeToPage property Makes the printed data grid contain only full rows.

nextPage() method Gets the next printable page of data.

validNextPage property Is `true` if printing the data requires an additional page.

Using the sizeToPage attribute to format pages

A PrintDataGrid page consists of the rows that are visible in the control's current view. Suppose, for example, that a PrintDataGrid control has a height of 130 pixels. The total height of each row and header is 30 pixels, and the control's data provider has 10 rows. In this situation, the printed PrintDataGrid page contains only three complete data rows, plus the header. The `sizeToPage` property specifies whether to include a fourth and partial data row.

The `sizeToPage` property, which is `true` by default, causes the PrintDataGrid control to remove any partially visible or empty rows and to resize itself to include only complete rows in the current view. For the data grid described in the preceding paragraph, when this property is `true`, the DataGrid shrinks to show three complete data rows, and no incomplete rows; if the attribute is `false`, the grid includes a partial row at the bottom.

The following properties provide information on page sizing that are affected by the `sizeToPage` property:

Property	Description
`currentPageHeight`	Contains the height of the grid, in pixels, that results if the `sizeToPage` property is `true`. If the `sizeToPage` property is `true`, the `currentPageHeight` property equals the `height` property.
`originalHeight`	Contains the grid height that results if the `sizeToPage` property is `false`. If the `sizeToPage` property is `false`, the `originalHeight` property equals the `height` property.

In most applications, you leave the `sizeToPage` attribute at its default value (`true`), and use the `height` property to determine the grid height.

The `sizeToPage` property does *not* affect the way the page breaks when a single PrintDataGrid control page is longer than a print page. To print multipage data grids without splitting rows, you must divide the grid items into multiple views by using the `nextPage()` method, as described in the following section.

Using the nextPage() method and validNextPage property to print multiple pages

The `validNextPage` property is `true` if the PrintDataGrid control has data beyond the rows that fit on the current print page. You use it to determine whether you need to format and print an additional page.

The `nextPage()` method lets you page through the data provider contents by setting the first row of the PrintDataGrid control to be the data provider row that follows the last row of the previous PrintDataGrid page. In other words, the `nextPage()` method increases the grid's `verticalScrollPosition` property by the value of the grid's `rowCount` property.

The following code shows a loop that prints a grid using multiple pages, without having rows that span pages:

```
// Queue the first page.
printJob.addObject(thePrintView);
// While there are more pages, print them.
while (thePrintView.myDataGrid.validNextPage) {
    //Put the next page of data in the view.
    thePrintView.myDataGrid.nextPage();
    //Queue the additional page.
    printJob.addObject(thePrintView);
}
```

The section "Example: Printing with multipage PrintDataGrid controls" on page 2047 shows how to use the `nextPage()` method to print a report with a multipage data grid.

Updating the PrintDataGrid layout

When you use a PrintDataGrid control to print a single data grid across multiple pages, you queue each page of the grid individually. If your application customizes each page beyond simply using the `nextPage()` method to page through the PrintDataGrid, you must call the `validateNow()` method to update the page layout before you print each page, as shown in "Print output component" on page 2050.

Example: Printing with multipage PrintDataGrid controls

The following example prints a data grid in which you can specify the number of items in the data provider. You can, therefore, set the DataGrid control contents to print on one, two, or more pages, so that you can see the effects of different-sized data sets on the printed result.

The example also shows how you can put header information before the grid and footer information after the grid, as in a shipping list or receipt. It uses the technique of selectively showing and hiding the header and footer, depending on the page being printed. To keep the code as short as possible, the example uses simple placeholder information only.

The application consists of the following files:

- The application file displays the form to the user, including TextArea and Button controls to set the number of lines and a Print button. The file includes the code to initialize the view, get the data, and handle the user's print request. It uses the FormPrintView MXML component as a template for the printed output.
- The FormPrintView.mxml file formats the printed output. It has two major elements:
  - The print output template includes the PrintDataGrid control and uses two MXML components to format the header and footer contents.
  - The `showPage()` function determines which sections of the template to include in a particular page of the output, based on the page type: first, middle, last, or single. On the first page of multipage output, the `showPage()` function hides the footer; on the middle and last pages, it hides the header. On a single page, it shows both header and footer.
- The FormPrintHeader.mxml and formPrintFooter.mxml files specify the contents of the start and the end of the output. To keep the application simple, the header has a single, static Label control. The footer displays a total of the numbers in the Quantity column. In a more complete application, the header page could have, for example, a shipping address, and the footer page could show more detail about the shipment totals.

The files include detailed comments explaining the purpose of the code.

Multipage print application file

The following code shows the multipage print application file:

```
<?xml version="1.0"?>
<!-- printing\MultiPagePrint.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="initData();">
    <fx:Script>
    <![CDATA[
        import mx.printing.*;
        import mx.collections.ArrayCollection;
        // Import the MXML custom print view control.
        import myComponents.FormPrintView;
        // Declare variables and initialize simple variables.
        // The dgProvider ArrayCollection is the DataGrid data provider.
        // It must be bindable because you change its contents dynamically.
        [Bindable]
        public var dgProvider:ArrayCollection;
        public var footerHeight:Number = 20;
        public var prodIndex:Number;
        public var prodTotal:Number = 0;

        // Data initialization, called when the application initializes.
        public function initData():void {
            // Create the data provider for the DataGrid control.
            dgProvider = new ArrayCollection;
        }

        // Fill the dgProvider ArrayCollection with the specified items.
        public function setdgProvider(items:int):void {
            // First initialize the index and clear any existing data.
            prodIndex=1;
            dgProvider.removeAll();

            // Fill the ArrayCollection, and calculate a product total.
            // For simplicity, it increases the Index field value by
            // 1, and the Qty field by 7 for each item.
            for (var z:int=0; z<items; z++)
            {
                var prod1:Object = {};
                prod1.Qty = prodIndex * 7;
                prod1.Index = prodIndex++;
                prodTotal += prod1.Qty;
                dgProvider.addItem(prod1);
            }
        }
        // The function to print the output.
        public function doPrint():void {
            // Create a FlexPrintJob instance.
            var printJob:FlexPrintJob = new FlexPrintJob();

            // Start the print job.
            if (printJob.start()) {
                // Create a FormPrintView control
                // as a child of the application.
                var thePrintView:FormPrintView = new FormPrintView();
                addElement(thePrintView);
```

```
// Set the print view properties.
thePrintView.width=printJob.pageWidth;
thePrintView.height=printJob.pageHeight;
thePrintView.prodTotal = prodTotal;

// Set the data provider of the FormPrintView
// component's DataGrid to be the data provider of
// the displayed DataGrid.
thePrintView.myDataGrid.dataProvider =
    myDataGrid.dataProvider;

// Create a single-page image.
thePrintView.showPage("single");

// If the print image's DataGrid can hold all the
// data provider's rows, add the page to the print job.
if(!thePrintView.myDataGrid.validNextPage)
{
    printJob.addObject(thePrintView);
}
// Otherwise, the job requires multiple pages.
else
{
    // Create the first page and add it to the print job.
    thePrintView.showPage("first");
    printJob.addObject(thePrintView);
    thePrintView.pageNumber++;

    // Loop through the following code
    // until all pages are queued.
    while(true)
    {
        // Move the next page of data to the top of
        // the PrintDataGrid.
        thePrintView.myDataGrid.nextPage();
        // Try creating a last page.
        thePrintView.showPage("last");
        // If the page holds the remaining data, or if
        // the last page was completely filled by the last
        // grid data, queue it for printing.
        // Test if there is data for another
        // PrintDataGrid page.
        if(!thePrintView.myDataGrid.validNextPage)
        {
            // This is the last page;
            // queue it and exit the print loop.
            printJob.addObject(thePrintView);
            break;
        }
        else
        // This is not the last page. Queue a middle page.
        {
            thePrintView.showPage("middle");
            printJob.addObject(thePrintView);
            thePrintView.pageNumber++;
        }
    }
```

```
                }
                // All pages are queued; remove the FormPrintView
                // control to free memory.
                removeElement(thePrintView);
            }
            // Send the job to the printer.
            printJob.send();
        }
    ]]>
    </fx:Script>
    <!-- The form that appears on the user's system.-->
    <mx:Form id="myForm" width="80%">
        <mx:FormHeading label="Product Information"/>
                <mx:DataGrid id="myDataGrid" dataProvider="{dgProvider}">
                <mx:columns>
                    <mx:DataGridColumn dataField="Index"/>
                    <mx:DataGridColumn dataField="Qty"/>
                </mx:columns>
            </mx:DataGrid>
        <mx:Text width="100%"
            text="Specify the number of lines and click Fill Grid first.
            Then you can click Print."/>
        <mx:TextInput id="dataItems" text="35"/>
        <mx:HBox>
            <mx:Button id="setDP"
                label="Fill Grid"
                click="setdgProvider(int(dataItems.text));"/>
            <mx:Button id="printDG"
                label="Print"
                click="doPrint();"/>
        </mx:HBox>
    </mx:Form>
</s:Application>
```

Print output component

The following lines show the FormPrintView.mxml custom component file:

```
<?xml version="1.0"?>
<!-- printing\myComponents\FormPrintView.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*"
    backgroundColor="#FFFFFF"
    paddingTop="50" paddingBottom="50" paddingLeft="50">
    <fx:Script>
        <![CDATA[
            import mx.core.*

            // Declare and initialize the variables used in the component.
            // The application sets the actual prodTotal value.
            [Bindable]
            public var pageNumber:Number = 1;
            [Bindable]
            public var prodTotal:Number = 0;
            // Control the page contents by selectively hiding the header and
            // footer based on the page type.
            public function showPage(pageType:String):void {
                if(pageType == "first" || pageType == "middle") {
                    // Hide the footer.
                    footer.includeInLayout=false;
                    footer.visible = false;
                }
                if(pageType == "middle" || pageType == "last") {
                    // The header won't be used again; hide it.
                    header.includeInLayout=false;
                    header.visible = false;
                }
                if(pageType == "last") {
                    // Show the footer.
                    footer.includeInLayout=true;
                    footer.visible = true;
                }
                //Update the DataGrid layout to reflect the results.
                validateNow();
            }
        ]]>
    </fx:Script>
    <!-- The template for the printed page,
        with the contents for all pages. -->
    <mx:VBox width="80%" horizontalAlign="left">
        <mx:Label text="Page {pageNumber}"/>
    </mx:VBox>
    <MyComp:FormPrintHeader id="header"/>

    <!-- The sizeToPage property is true by default, so the last
        page has only as many grid rows as are needed for the data. -->
    <mx:PrintDataGrid id="myDataGrid" width="60%" height="100%">
    <!-- Specify the columns to ensure that their order is correct. -->
        <mx:columns>
            <mx:DataGridColumn dataField="Index" />
            <mx:DataGridColumn dataField="Qty" />
        </mx:columns>
    </mx:PrintDataGrid>

    <!-- Create a FormPrintFooter control
        and set its prodTotal variable. -->
    <MyComp:FormPrintFooter id="footer" pTotal="{prodTotal}"/>
</mx:VBox>
```

Header and footer files

The following lines show the FormPrintHeader.mxml file.

```
<?xml version="1.0"?>
<!-- printing\myComponents\FormPrintHeader.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="60%"
    horizontalAlign="right" >
    <mx:Label text="This is a placeholder for first page contents"/>
</mx:VBox>
```

The following lines show the FormPrintFooter.mxml file:

```
<?xml version="1.0"?>
<!-- printing\myComponents\FormPrintFooter.mxml -->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="60%"
    horizontalAlign="right">

    <!-- Declare and initialize the product total variable. -->
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var pTotal:Number = 0;
        ]]>
        </fx:Script>
    <mx:Label text="Product Total: {pTotal}"/>
</mx:VBox>
```

Using the PrintAdvancedDataGrid control

The PrintAdvancedDataGrid and PrintOLAPDataGrid controls provide the same functionality for the AdvancedDataGrid and OLAPDataGrid controls as the PrintDataGrid control does for the DataGrid control. For more information, see “Using the PrintDataGrid control for multipage grids” on page 2045.

The following example uses the PrintAdvancedDataGrid control to print an instance of the AdvancedDataGrid control.

```
<?xml version="1.0"?>
<!-- printing\ADGPrint.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.printing.*;
            import mx.collections.ArrayCollection;
            import mx.printing.PrintAdvancedDataGrid;

            include "SimpleHierarchicalData.as";
            // Create a PrintJob instance.
            private function doPrint():void {
                // Create an instance of the FlexPrintJob class.
                var printJob:FlexPrintJob = new FlexPrintJob();

                // Initialize the PrintAdvancedDataGrid control.
                var printADG:PrintAdvancedDataGrid =
                    new PrintAdvancedDataGrid();
                // Exclude the PrintAdvancedDataGrid control from layout.
                printADG.includeInLayout = false;
                printADG.source = adg;
                // Add the print-specific control to the application.
                addElement(printADG);

                // Start the print job.
                if (printJob.start() == false) {
                    // User cancelled print job.
                    // Remove the print-specific control to free memory.
                    removeElement(printADG);
                    return;
                }
                // Add the object to print. Do not scale it.
                printJob.addObject(printADG, FlexPrintJobScaleType.NONE);
                // Send the job to the printer.
                printJob.send();
                // Remove the print-specific control to free memory.
                removeElement(printADG);
            }
        ]]>
    </fx:Script>
    <mx:VBox id="myVBox"
        width="100%" height="100%">
        <mx:AdvancedDataGrid id="adg"
            width="100%" height="100%">
            <mx:dataProvider>
                <mx:HierarchicalData source="{dpHierarchy}"/>
            </mx:dataProvider>
            <mx:columns>
                <mx:AdvancedDataGridColumn dataField="Region"/>
                <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                    headerText="Territory Rep"/>
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:columns>
        </mx:AdvancedDataGrid>
        <mx:Button id="myButton"
            label="Print"
            click="doPrint();"/>
    </mx:VBox>
</s:Application>
```

The contents of the included SimpleHierarchicalData.as file are as follows:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", children: [
     {Region:"Arizona", children: [
        {Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
        {Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]},
     {Region:"Central California", children: [
        {Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]},
     {Region:"Nevada", children: [
        {Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]},
     {Region:"Northern California", children: [
        {Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
        {Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]},
     {Region:"Southern California", children: [
        {Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
        {Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
  ]}
]);
```

This example uses the `PrintAdvancedDataGrid.source` property to initialize the PrintAdvancedDataGrid control from the AdvancedDataGrid control.

To support the AdvancedDataGrid control, the PrintAdvancedDataGrid control adds the following properties not available in the PrintDataGrid control:

Property	**Description**
`allowInteractions`	If `true`, allow some interactions with the control, such as column resizing, column reordering, and expanding or collapsing nodes. The default value is `false`.

Property	**Description**
`displayIcons`	If `true`, display the folder and leaf icons in the navigation tree. The default value is `true`.
`source`	Initialize the PrintAdvancedDataGrid control and all of its properties from the specified AdvancedDataGrid control.
`validPreviousPage`	Indicates that the data provider contains data rows that precede the rows that the PrintAdvancedDataGrid control currently displays.

Localization

Flex includes robust support for localizing your applications.

Introduction to localization

Localization is the process of including assets and formatting an application to support a locale. A *locale* is the combination of a language and a country code; for example, en_US refers to the English language as spoken in the United States, and fr_FR refers to the French language as spoken in France. To localize an application for the US and France, you would provide two sets of assets, one for the en_US locale and one for the fr_FR locale.

Locales can share languages. For example, en_US and en_GB (Great Britain) are different locales and therefore use different sets of assets. In this case, both locales use the English language, but the country code indicates that they are different locales, and might therefor use different assets. For example, an application in the en_US locale might spell the word “color”, whereas the word would be “colour” in the en_GB locale. Also, units of currency would be represented in dollars or pounds, depending on the locale, and the format of dates and times would also be different.

Localization goes beyond formatting numbers and translating strings that are used in your application. It can also include any type of asset such as audio files, images, styles, SWF files, and video files.

There are two ways to localize an application:

- Using built-in localization support. Flex includes localization-aware classes for the following:
  - Currency formatters
  - Number formatters
  - Date and time formatters
  - Layout mirroring
  - Bidirectional text
  - Sorters and collators
- Using resource bundles. Resource bundles let you define assets such as strings and images for various locales and use those assets at run time.

You can use a combination of these approaches, depending on how much of your application needs to be localized.

More Help topics

The flash.globalization package in Flash Player: Cultural diversity without complexity

“Resource Bundles” on page 2091

Localization in Flex – Part 1: Compiling resources into an application

Localization in Flex – Part 2: Loading resources at runtime

Library containing utility classes for sorting application locales

Setting the locale

When an application starts up, you should generally determine which locale the user wants and then set the application's `locale` style property to that locale. This ensures that the localization-aware classes such as formatters and sorters behave as the user expects. If you use resource bundles, then you should use some method to determine the user's preferred locale, and load the appropriate resource bundle.

To determine which locale your users want to use, you can use one or more of the following methods:

- User prompt — You can start the application in a default locale, and then ask the user to choose their preferred locale from a list of pre-determined locales.
- `Capabilities.language` — The `Capabilities.language` property in ActionScript provides the language code for Adobe® Flash® Player and Adobe AIR™. On English systems, this property returns only the language code, not the country code. This property returns the user interface language, which refers to the language used for all menus, dialog boxes, error messages, and help files in Flash Player and AIR.
- `Accept-Language` header — When a browser makes an HTTP request, it sends a list of languages to the server in the `Accept-Language` header. You can parse this header in your HTML wrapper and pass it as a `flashVars` variable into your Flex application. If you use URLLoader to load an application, you will not have access to this header.
- Browser preferences — You can access the browser's language settings in the HTML wrapper by using JavaScript. You can then pass the values as `flashVars` variables to your application. You can also use the ExternalInterface API to access the values from within your Flex application. The language is typically accessible via the Navigator object's `language` property. Depending on the browser, you also access the `userLanguage` or `systemLanguage` properties. For more information on using the ExternalInterface API, see "Using the ExternalInterface API to access JavaScript" on page 235.

Built-in localization support

Typically, you set the `locale` style property of the application or the component to the value of the `LocaleID.DEFAULT` property. This instructs Flex to use the client's operating system for locale information (such as the currency symbol, date/time format, or preferred sorting rules).

You can set the `locale` style property on an individual component or on the entire application. The style is inheritable, which means if you set it on the application, then all components in that application inherit that style, unless they specifically override it. If you do not set the `locale` style property, the compiler uses the global defaults that are defined in the defaults.css style sheet. In this case, the default value is "en".

The following example sets the value of the application's `locale` style property to the `LocaleID.DEFAULT` constant. When you run this example, the value of the DateTimeFormatter's `actualLocaleIDName` property will be whatever language preference you have set in your operating system.

```
<?xml version="1.0"?>
<!-- l10n/LocaleApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Declarations>
        <s:DateTimeFormatter id="dateTimeFormatter"/>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
        import flash.globalization.LocaleID;

        public function initApp():void {
            this.setStyle("locale", LocaleID.DEFAULT);
        }

        public function doSomething():void {
            l1.text = dateTimeFormatter.actualLocaleIDName;
        }
        ]]>
    </fx:Script>
    <s:VGroup>
        <s:Label id="l1"/>
        <s:Button id="b1" label="Click Me" click="doSomething()"/>
    </s:VGroup>

</s:Application>
```

You can create a LocaleID object to retrieve data about the locale. You can use methods such as `getKeysAndValues()`, `getLanguage()`, `getRegion()`, `getScript()`, `getVariant()`, and `isRightToLeft()` to set properties and create other objects in your application.

Resource bundles

For applications that use resource bundles, you must explicitly compile assets for the supported locales into the application. You can then let your users choose from a list of the compiled-in locales. After determining the user's preferred locale, you set the value of the ResourceManager's `localeChain` property so that that locale's assets are used in your application.

Formatting dates, numbers, and currencies

Flex includes the following Spark classes for formatting dates, numbers and currencies:

- spark.formatters.CurrencyFormatter
- spark.formatters.NumberFormatter
- spark.formatters.DateTimeFormatter

Each of these classes is based on a similar class in the flash.globalization.* package. Flex lets you instantiate these formatters in MXML as well as ActionScript.

The formatters' output might differ depending on the runtime environment of the client. The formatters call the underlying operating system which can return different results, depending on the device.

The Spark formatters inherit the `locale` style property, have bindable properties and methods, and generate events.

As with other non-visual classes, you declare formatters in the `<fx:Declarations>` block of your applications.

The following example uses a CurrencyFormatter and a NumberFormatter to display formatted values in the client's default locale. The currency symbol and grouping separator (sometimes called the thousands separator) are defined by whatever the client machine's locale is. The client's preferred locale is obtained by using the `LocaleID.DEFAULT` property.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- l10n/CurrencyFormatterApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Get the client's default locale from the OS
            by using the LocaleID.DEFAULT constant. -->
        <s:CurrencyFormatter id="curFormatter"
            useCurrencySymbol="true"
            locale="{LocaleID.DEFAULT}"/>
        <s:NumberFormatter id="numFormatter"
            locale="{LocaleID.DEFAULT}"/>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import flash.globalization.LocaleID;
        ]]>
    </fx:Script>

    <s:Label id="la1" text="{curFormatter.format(4500000)}"/>
    <s:Label id="la2" text="{numFormatter.format(4500000)}"/>

</s:Application>
```

You can explicitly let a user select their locale, and bind your formatters to the selected locale. You can also create a new LocaleID object based on the selected locale so that you can get other details about the locale.

The following example lets a user select from a list of locales, and then returns various information about that locale, including the script, language, and currency symbol:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- l10n/CurrencyFormatterSelector.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600">
    <fx:Declarations>
        <s:CurrencyFormatter id="curFormatter"
            useCurrencySymbol="true"
            locale="{myDDL.selectedItem}"/>
    </fx:Declarations>

    <fx:Script>
        import flash.globalization.LocaleID;

        private function updateLocaleDetails():void {
            var locale:LocaleID = new LocaleID(myDDL.selectedItem);
            l6.text = "Locale Script: " + locale.getScript();
            l7.text = "Locale Region: " + locale.getRegion();
            l8.text = "Locale Language: " + locale.getLanguage();
        }
    </fx:Script>

    <s:VGroup>
        <s:HGroup>
            <s:Label text="Select Locale ID: "/>
            <s:DropDownList id="myDDL"
                change="updateLocaleDetails()"
                width="200"
                prompt="Select One"
                labelField="product">
                <mx:ArrayCollection>
                    <fx:String>ar-SA</fx:String>
                    <fx:String>da-DK</fx:String>
                    <fx:String>de-DE</fx:String>
                    <fx:String>en-GB</fx:String>
                    <fx:String>en-US</fx:String>
                    <fx:String>es-ES</fx:String>
                    <fx:String>fi-FI</fx:String>
                    <fx:String>fr-FR</fx:String>
                    <fx:String>it-IT</fx:String>
                    <fx:String>ja-JP</fx:String>
                    <fx:String>ko-KR</fx:String>
                    <fx:String>nb-NO</fx:String>
                    <fx:String>nl-NL</fx:String>
                    <fx:String>pt-BR</fx:String>
                    <fx:String>ru-RU</fx:String>
                    <fx:String>sv-SE</fx:String>
                    <fx:String>zh-CN</fx:String>
                    <fx:String>zh-TW</fx:String>
                </mx:ArrayCollection>
            </s:DropDownList>
        </s:HGroup>
        <s:Label id="currencyLabel"
            fontWeight="bold"
            fontSize="16"
            text="{curFormatter.format(4500000)}"/>
        <s:Label id="l2" text="Actual Locale ID Name: {curFormatter.actualLocaleIDName}"/>
        <s:Label id="l4" text="Currency Symbol: {curFormatter.currencySymbol}"/>
        <s:Label id="l5" text="Currency ISO Code: {curFormatter.currencyISOCode}"/>
        <s:Label id="l6" text="Locale Script: "/>
        <s:Label id="l7" text="Locale Region: "/>
        <s:Label id="l8" text="Locale Language: "/>
        <s:Label text="All Available Locale ID Names: "/>
        <s:TextArea id="ta1"
            text="{String(CurrencyFormatter.getAvailableLocaleIDNames())}"
            width="300" height="400"/>
    </s:VGroup>
</s:Application>
```

In addition, this example shows a list of all available locales. These are the locales that are supported by the client's operating system.

In some cases, you do not necessarily want your application to support all possible locales, or you want your application to use only one locale. In these cases, you can set the locale(s) programmatically. To do this, you can use the `locale` style property to apply a particular locale to individual components or to an entire application; for example:

```
<fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    s|Application {
        locale: 'fr-FR';
    }
</fx:Style>
```

You can also use resource bundles to configure the formatters. For more information, see "Custom formatting with resource bundles" on page 2119.

For more information on using the Spark formatter classes, see "Formatting Data" on page 2004.

Mirroring and bidirectional text

One aspect of localizing applications is the direction of the text. Some languages are read from left to right, such as English, and other languages are read from right to left, such as Hebrew.

The layout of the controls and the direction of text inputs should match the direction of the language. For example, an application that uses an RTL language should layout on the right side of the application's stage.

There are two primary ways to control the appearance of an application that supports both left-to-right (LTR) and right-to-left (RTL) languages:

- Layout mirroring — Layout mirroring is the process of flipping the appearance of the layout controls. For example, in an RTL application, vertical scroll bars should appear on the left instead of the right. For horizontal scrollbars, the thumb should appear on the right side of the control instead of the left. Layout mirroring is commonly used to make a UI match the direction of an RTL language.

- Text direction — Text direction defines the direction that text flows in the text controls. For RTL languages such as Hebrew and Arabic, the characters should render from the right side to the left. This also includes bidirectional text support, where a control might need to render some of the text RTL, and some of the text LTR in the same control. The Spark text controls handle directionality of the text internally. They do not use mirroring to render text.

To change an application's layout from LTR to RTL, set the `layoutDirection` property of the application object to `"rtl"`. The direction of text is automatically set based on the characters used, but you should also set the `direction` style property to `"rtl"` if the majority of your text is RTL. This property instructs the text control to apply RTL rules to the text.

For more information, see "Mirroring and bidirectional text" on page 2067.

Transforming text

The spark.globalization.StringTools class provides methods to convert letters to minuscules (lowercase letters) or majuscules (uppercase letters) for languages based on writing systems that incorporate the concept of capitalization (essentially, the Armenian, Latin, Cyrillic, and Greek alphabets). Using the StringTools class enables your application to use language-specific rules to perform those transformations.

The Spark StringTools class is based on the flash.globalization.StringTools class. This lets you use this class in MXML as well as ActionScript, and also lets it inherit the value of the `locale` style property. In addition, the Spark StringTools class has bindable properties and methods and generates events.

As with formatters, you declare an instance of the StringTools class in the `<fx:Declarations>` block of your application.

The following example uses the StringTools class to convert a string to all lower case:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- l10n/TransformingText.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:StringTools id="st1" locale="{LocaleID.DEFAULT}"/>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import flash.globalization.LocaleID;
        ]]>
    </fx:Script>
    <s:Label id="l1" text="{st1.toLowerCase('THIS LABEL WAS IN ALL CAPS.')}"/>
</s:Application>
```

Note that some characters for some languages do not have the same character length before and after case conversion. For example, the lower case sharp-s ("ß") is represented with two characters, "SS", when converted to upper case in the German language.

Sorting and matching

When ordering a list that is presented visually to the user, it is essential to follow the user's expectations. It is not just a question of aesthetics: expecting it at the end of the list, a Swedish user might fail to find an element that begins with "ö" if placed next to the plain "o". Fortunately, avoiding that problem is easy: sort the list with a Sort or SortingCollator class that is configured for the user's locale.

The sorting and matching classes' output might differ depending on the runtime environment of the client. The collators call the underlying operating system which can return different results, depending on the device.

As with formatters, the Spark Sort and SortField classes inherit the value of the `locale` style property on the application, or you can set the locale style on the component.

The following example creates a Spark Sort with two Spark SortFields. Select a locale from the the drop-down list to change the sort order to match the preferences of the selected locale.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- l10n/SortLocale2.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600">
    <fx:Declarations>
        <s:Sort id="sortbyLastName_FirstName">
            <s:fields>
                <s:SortField name="last"/>
                <s:SortField name="first"/>
            </s:fields>
        </s:Sort>
        <mx:ArrayCollection id="collection" sort="{sortbyLastName_FirstName}">
            <mx:source>
                <fx:Object first="Anders" last="Öhlund"/>
                <fx:Object first="Eileen" last="Oehland"/>
                <fx:Object first="Aiden" last="Zorn"/>
                <fx:Object first="Steve" last="Ohlin"/>
            </mx:source>
        </mx:ArrayCollection>
    </fx:Declarations>

    <fx:Script>
        /* Sets the locale style on the document UI component.
            The SortField and Sort objects defined in the
            fx:Declarations section will inherit this style. */
        private function updateSort(e:Event):void {
            setStyle('locale', myDDL.selectedItem);
            collection.refresh();
        }
    </fx:Script>

    <s:VGroup>
        <s:HGroup>
            <s:Label text="Select Locale ID Name: "/>
            <s:DropDownList id="myDDL" prompt="Select One"
                width="200"
                labelField="product"
                change="updateSort(event)">
                <mx:ArrayCollection>
                    <fx:String>en-US</fx:String>
                    <fx:String>sv-SE</fx:String>
                    <fx:String>zh-CN</fx:String>
                </mx:ArrayCollection>
            </s:DropDownList>
        </s:HGroup>
        <s:DataGrid
            dataProvider="{collection}"
            creationComplete="{collection.refresh()}">
            <s:columns>
                <s:ArrayList>
                    <s:GridColumn dataField="last"/>
                    <s:GridColumn dataField="first"/>
                </s:ArrayList>
            </s:columns>
        </s:DataGrid>
    </s:VGroup>
</s:Application>
```

The following example uses the Spark Sort and SortField classes in ActionScript to create a locale-dependent sort:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- l10n/SweeterSwedishSort.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="500"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.collections.ISort;
            import mx.collections.ISortField;
            import mx.collections.ArrayCollection;
            import spark.collections.Sort;
            import spark.collections.SortField;

            private var defaultSort:ISort = new Sort();
            private var swedishSort:ISort = new Sort();
            private var a:Array = new Array('Öhlund','Oehland','Zorn','Ohlin');
            private var names:ArrayCollection = new ArrayCollection(a);
            private var sortField:SortField = new SortField(null);
            private function initApp():void {
                originalOrder.text = "Original Order: " + names.toString();
                /* Add a style client, so sortField can pick up the locale style. */
                addStyleClient(sortField as spark.collections.SortField);
                /* Add the sortField to the fields array of the sorts. */
                defaultSort.fields = [sortField];
                swedishSort.fields = [sortField];
            }
            private function changeSortOrder(s:String):void {
                if (s=="en-US") {
                    /* Set the locale for the SortField. */
                    sortField.setStyle("locale","en-US");
                    names.sort = defaultSort;
                    /* Refresh the ArrayCollection after changing the sort property. */
                    names.refresh();
                    ta1.text = "Default Order: " + names.toString();
                } else {
                    /* Set the locale for the SortField. */
                    sortField.setStyle("locale","sv-SE");
                    names.sort = swedishSort;
                    /* Refresh the ArrayCollection after changing the sort property. */
                    names.refresh();
                    ta1.text = "Swedish Order: " + names.toString();
                }
            }
        ]]>
    </fx:Script>

    <s:Label id="originalOrder" paddingTop="10" fontWeight="bold"/>
    <s:TextArea id="ta1" height="200" width="400"/>
    <s:HGroup>
        <s:Button label="Show Default Sort Order" click="changeSortOrder('en-US')"/>
        <s:Button label="Show Swedish Sort Order" click="changeSortOrder('sv-SE')"/>
    </s:HGroup>
</s:Application>
```

The following example shows a custom sort that also depends on locale. This example uses the SortingCollator class and defines the logic for the sort. As with formatters and StringTools, you declare collators in the `<fx:Declarations>` block your application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- l10n/SwedishSort.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600"
               creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:SortingCollator id="defaultSorter" locale="{LocaleID.DEFAULT}"/>
        <s:SortingCollator id="swedishSorter" locale="sv-SE"/>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import flash.globalization.LocaleID;
            import mx.collections.ArrayCollection;

            private var a:Array = new Array('Öhlund','Oehland','Zorn','Ohlin');
            [Bindable]
            private var names:ArrayCollection = new ArrayCollection(a);

            private function initApp():void {
                originalOrder.text = "Original Order: " + names.toString();
                sortNames(defaultSorter);
                label1.text = "Default Order: " + names.toString();
                sortNames(swedishSorter);
                label2.text = "Swedish Order: " + names.toString();
            }

            private function sortNames(sorter:SortingCollator):void {
                var changed:Boolean = false;
                while (!changed) {
                    changed = true;
                    for (var i:int = 0; i < names.length - 1; i++) {
                        var comparisonResult:int = sorter.compare(names[i],names[i+1]);
                        if (comparisonResult > 0) {
                            var s:String = names[i];
                            names[i] = names[i + 1];
                            names[i + 1] = s;
                            changed = false;
                        }
                    }
                }
            }
        ]]>
    </fx:Script>
    <s:Label id="originalOrder" paddingTop="10" fontWeight="bold"/>
    <s:Label id="label1"/>
    <s:Label id="label2"/>
</s:Application>
```

Matching is typically performed when the user searches for an item (for example, searching for a word in a document). When matching, you might want to ignore minor differences to increase the chances of finding relevant matches. For example, capitalization is commonly ignored. Diacritical marks are also often disregarded because not all users enter them as expected (either because their keyboard layout makes it difficult, or as the result of a spelling mistake).

You use the `<s:MatchingCollator>` tag to configure your matching rules. The following example shows how the locale and settings on the MatchingCollator can affect string comparisons:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- l10n/MatchingCollatorExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:MatchingCollator id="matchingCollator"
            ignoreCase="false"
            ignoreCharacterWidth="false"
            ignoreDiacritics="{cb1.selected}"
            ignoreKanaType="false"
            ignoreSymbols="false"
            numericComparison="false"
            locale="{myDDL.selectedItem}"/>
    </fx:Declarations>

    <fx:Script>
```

```
        <![CDATA[
            private function doCompares():void {
                label1.text = matchingCollator.compare('ä','vu').toString();
                label2.text = matchingCollator.compare('a','ä').toString();
                label3.text = matchingCollator.compare('coté','côte').toString();
            }
        ]]>
    </fx:Script>
    <s:HGroup>
        <s:Label text="Select Locale ID: "/>
        <s:DropDownList id="myDDL"
            change="doCompares()"
            width="200"
            prompt="Select One">
            <mx:ArrayCollection>
                <fx:String>en-US</fx:String>
                <fx:String>fr-FR</fx:String>
            </mx:ArrayCollection>
        </s:DropDownList>
        <s:CheckBox id="cb1"
            label="Ignore Diacritics"
            change="doCompares()"/>
    </s:HGroup>
    <s:Form>
        <s:FormItem label="{matchingCollator.actualLocaleIDName}"/>
        <s:FormItem label="String1: ä, String2: vu">
            <s:Label id="label1" text=""/>
        </s:FormItem>
        <s:FormItem label="String1: a, String2: ä">
            <s:Label id="label2" text=""/>
        </s:FormItem>
        <s:FormItem label="String1: coté, String2: côte">
            <s:Label id="label3" text=""/>
        </s:FormItem>
    </s:Form>
    <s:Label text="-1: string1 is less than string 2; 0: strings are equal; 1: string1 is 
greater than string2"/>
</s:Application>
```

Additional collator examples are available on the Adobe Globalization blog.

Resource bundles

A *resource bundle* is a list of assets that is organized by locale. These assets can be just about anything, from strings, numbers, formats, and images to styles. You typically have a separate resource bundle for each locale that your application supports.

To localize an application with resource bundles, you first create properties files that define the localized assets. You then compile these properties files into the application as resource bundles, or create resource modules from the properties files and load them at run time. To compile resources into an application, you pass a comma-separated list of locales to the the `locale` compiler option. This compiler option has no effect on the spark.globalization.* classes.

You can also load pre-compiled resource modules and define resource bundles at run time. For more information on creating and using resource bundles in your applications, see “Resource Bundles” on page 2091.

Mirroring and bidirectional text

Introduction to mirroring and bidirectional text

here are two primary ways to control the appearance of an application that supports both left-to-right (LTR) and right-to-left (RTL) languages:

- Layout mirroring — Layout mirroring is the process of flipping the appearance of the layout controls. For example, in an RTL application, vertical scroll bars should appear on the left instead of the right. For horizontal scrollbars, the thumb should appear on the right side of the control instead of the left. Layout mirroring is commonly used to make a UI match the direction of an RTL language.
- Text direction — Text direction defines the direction that text flows in the text controls. For RTL languages such as Hebrew and Arabic, the characters should render from the right side to the left. This also includes bidirectional text support, where a control might need to render some of the text RTL, and some of the text LTR in the same control. The Spark Flex text controls handle directionality of the text internally. They do not use mirroring to render text.

To change an application's layout from LTR to RTL, set the `layoutDirection` property of the application object to "rtl". The direction of text is automatically set based on the characters used, but you should also set the `direction` style property to "rtl" if the majority of your text is RTL. This property instructs the text control to apply RTL rules to the text.

The following example lets you change the `layoutDirection` and `direction` style properties on the application from "ltr" to "rtl":

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring/DynamicMirroring.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               height="400" width="400">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.core.LayoutDirection;

            private function setDirectionProps():void {
                panel1.setStyle('direction', ddl1.selectedItem);
                panel1.setStyle('layoutDirection', ddl1.selectedItem);
            }
        ]]>
    </fx:Script>

    <s:Panel id="panel1" title="Panel With Non-Mirrored Content" width="350"
             layoutDirection="{LayoutDirection.LTR}">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Label text="This panel contains content that is not mirrored."/>
        <s:Button label="This Button is not mirrored in the container"/>
        <s:DropDownList id="ddl1" requireSelection="true" selectedIndex="0"
change="setDirectionProps()">
            <s:ArrayList source="['ltr','rtl']"/>
        </s:DropDownList>
    </s:Panel>
</s:Application>
```

Mirroring is not supported when you set the `compatibility-version` compiler option to 3.0.0 or when the theme is Halo or Mobile. Mirroring is only supported by the Wireframe and Spark themes.

More Help topics

Flex mirroring specification

W3C site on RTL/Internationalization

Description of the Unicode bidirectional algorithm

Google transliteration

"Localization" on page 2055

Layout mirroring

Layout mirroring refers to how containers and controls are drawn on the screen. The default direction is LTR. You can change any container or control that implements the ILayoutDirectionElement interface to lay out from right to left by setting the `layoutDirection` property to "rtl". This interface includes all controls that implement the IVisualElement interface as well as some assets that do not implement this interface.

The recommended method of using `layoutDirection` is to set it on the Application container so that all child containers and controls in the application inherit its value. You can set the `layoutDirection` property in one of the following ways:

- In MXML. For example:

  ```
  <s:Panel id="myPanel" layoutDirection="rtl">
  ```

- With the `setStyle()` method. For example:

  ```
  myPanel.setStyle("layoutDirection","rtl");
  ```

The following example shows two Panel containers. One sets the `layoutDirection` property to "rtl" and the other to "ltr":

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring/SimpleLayoutMirroring.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               height="400" width="400">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.core.LayoutDirection;
        ]]>
    </fx:Script>

    <s:Panel title="Panel With Non-Mirrored Content" width="350"
             layoutDirection="{LayoutDirection.LTR}">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Label text="This panel contains content that is not mirrored."/>
        <s:Button label="This Button is not mirrored in the container"/>
        <s:DropDownList requireSelection="true" selectedIndex="0">
            <s:ArrayList source="['Drop','Down','List']" />
        </s:DropDownList>
    </s:Panel>

    <s:Panel title="Panel With Mirrored Content" width="350"
             layoutDirection="{LayoutDirection.RTL}">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Label text="This panel contains content that is mirrored."/>
        <s:Button label="This Button is mirrored inside the container"/>
        <s:DropDownList requireSelection="true" selectedIndex="0">
            <s:ArrayList source="['Drop','Down','List']" />
        </s:DropDownList>
    </s:Panel>
</s:Application>
```

Instead of using the strings "ltr" or "rtl", the `layoutDirection` style property can also take the following constants:

- `mx.core.LayoutDirection.LTR`
- `mx.core.LayoutDirection.RTL`

Text direction

Text direction refers to the direction that characters are rendered in text-based controls. Some languages, such as Hebrew and Arabic, are read from right to left. Most text-based controls auto-detect the direction of the text based on the character codes of the text content, but specifying the direction ensures that the punctuation follows the appropriate (RTL or LTR) rules.

As with the `layoutDirection` property, you typically set the `direction` property on the application or the parent container of the text-based control.

You can set the `direction` property in one of the following ways:

- In MXML. For example:

```
<s:Panel id="myPanel" direction="rtl">
```

- With the `setStyle()` method. For example:

```
myPanel.setStyle("direction","rtl");
```

The following example shows two Panel containers. One sets the `direction` property to RTL and the other to LTR:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring/SimpleTextMirroring.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               height="400" width="400">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.core.LayoutDirection;
        ]]>
    </fx:Script>

    <s:Panel title="Panel With Non-Mirrored Content" width="350"
             direction="ltr">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Label text="This panel contains content that is not mirrored."/>
        <s:Button label="This Button is not mirrored in the container."/>
        <s:DropDownList requireSelection="true" selectedIndex="0">
            <s:ArrayList source="['Drop','Down','List']" />
        </s:DropDownList>
    </s:Panel>

    <s:Panel title="Panel With Mirrored Content" width="350"
             direction="rtl">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:Label text="This panel contains content that is mirrored."/>
        <s:Button label="This Button is mirrored inside the container."/>
        <s:DropDownList requireSelection="true" selectedIndex="0">
            <s:ArrayList source="['Drop','Down','List']" />
        </s:DropDownList>
    </s:Panel>
</s:Application>
```

The `direction` style property only affects punctuation in this example. The characters themselves are not rendered from right to left because they use an LTR-based character set.

Support for setting the direction on text controls is built into text controls in the Spark component set because those text controls are based on FTE (Flash Text Engine). Text-based controls that are based on the TextField control do not support mirroring or bidirectional text. To use FTE with MX text-based controls, you must adjust your compiler settings. For more information, see "Mirroring with text controls" on page 2077.

Bidirectional text refers to text that contains both RTL and LTR content. For example, if you write a sentence in an RTL language such as Hebrew, but include the name of a company, such as Adobe. The compiler recognizes characters codes that use RTL and LTR and renders the text appropriately.

The following example shows bidirectional text:

```
<?xml version="1.0"?>
<!-- mirroring/BiDiText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="addText()">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flashx.textLayout.formats.*;
            import spark.utils.TextFlowUtil;
            private function addText():void {
                myRT.textFlow = TextFlowUtil.importFromString("school is written " +
                    String.fromCharCode(0x0645, 0x062f, 0x0631, 0x0633, 0x0629) +
                    " in Arabic and " + String.fromCharCode(0x05E1, 0x05B5, 0x05E4, 0x05B6,
0x05E8) +
                    " in Hebrew.");
            }
        ]]>
    </fx:Script>

    <s:Panel title="Example of Bidirectional text">
        <s:RichText id="myRT" width="400" height="150"/>
    </s:Panel>
</s:Application>
```

In this example, the `direction` property is set to "rtl". This causes the punctuation to be reversed so that the period is on the left.

In some cases, you cannot set the `direction` style in MXML because some components (such as the ProgressBar and TileList controls) already define a `direction` property that defines other behavior. For more information, see "Overlapping direction properties" on page 2090.

Inheritance

The values of the `layoutDirection` and `direction` style properties are inherited by most components from their parent. As a result, you typically set these properties on the Application tag so that the properties are used consistently across the entire application. While it is possible to set these styles on individual components, this can be a cumbersome and mistake-prone approach.

In addition, if you set the style properties on individual components, you override the defaults set in the global.css file. These defaults define expected behavior. For example, you typically do not want to reverse the direction of an Image or VideoDisplay control when you mirror the contents of an application.

Some components, such as text-based controls, and the Image, BitmapImage, VideoPlayer, and VideoDisplay controls do not inherit the value of the `layoutDirection` style property. These components override the values of this property in the defaults.css file. They either do not support the property, support it in a different manner, or do not inherit the value because inheriting it would create unexpected results.

The following table describes the components that do not inherit the value of the `layoutDirection` style property:

Control	Description
BitmapImage, Image	The Image (both Spark and MX) and BitmapImage components do not inherit the value of the `layoutDirection` property from the container because Flex assumes that you do not want to mirror the contents. To override this behavior and reverse the contents of the MX Image and Spark BitmapImage controls, set the value of the `layoutDirection` property directly on the control. You cannot reverse the contents of the Spark Image control with the `layoutDirection` property. For more information, see “Images and video controls” on page 2081.
VideoDisplay, VideoPlayer	The VideoDisplay and VideoPlayer components do not inherit the value of the `layoutDirection` property for the same reason that the Image components do not: the assumption is that the contents should not be mirrored. However, you can override the default behavior by setting the `layoutDirection` property directly on the control. In this case, the VideoDisplay control mirrors the actual video being displayed. The VideoPlayer mirrors the subcontrols that are used to control the video but not the video itself.
Spark text-based controls, such as those based on the TextBase class (Label and RichText) and those including the RichEditableText control (RichEditableText, TextArea, and TextInput)	The reason that the Spark text-based controls do not inherit the value of the `layoutDirection` property is that they use FTE/TLF to auto-detect the direction of the text. The text characters are not mirrored in the same way that UI controls are mirrored. The input cursor for the TextArea and TextInput controls does not rely on the `layoutDirection` property, but rather on the value of the direction property to determine the starting position of the cursor when the control gains focus.

In addition to being a style, the `layoutDirection` property is defined on the ILayoutDirectionElement interface. All UIComponents inherit the properties of this interface. As a result, you can also set the `layoutDirection` as a property on an object. If a component's `layoutDirection` property is set to `null`, the component inherits the value from its parent. For components that implement the ILayoutDirectionElement interface, this means that setting `layoutDirection=null` is equivalent to `setStyle("layoutDirection", undefined).`

Components

This section describes the behavior of the components when you set `layoutDirection` to "rtl" on the Application container. In most cases, the behavior of Spark and MX controls is the same.

Basic UI controls

Control	Description
Alert	The Alert title is on the right. The text is not mirrored. Any icons are on the right. All buttons are arranged the opposite of their LTR order. For example, if buttons were "OK" and "Cancel", they appear as "Cancel" and "OK".
Button	The appearance of the Button is mirrored but the label text is not. If you use the Button control with an embedded icon, this icon is aligned on the right but the image itself is not reversed.
CheckBox	The label is rendered on the left. The check icon is not mirrored.
ColorPicker	The layout of the selector box and text input are reversed. The color grid is reversed, with the most common colors appearing on the right.

Control	Description
DateChooser	The direction of days and months are mirrored. The text for the days and numbers is not mirrored. The forward month arrow is on the left and back month arrow button is on the right.
DateField	The calendar icon appears on the left with the text input on the right. The calendar that opens looks like a mirrored DateChooser control. When you place a DateField control, be sure to leave enough room on the left side for the popup. When there is sufficient room on the left the DateChooser popup is aligned correctly. If there is not enough area on the left, the calendar that pops up can be cut off by the screen edge.
DownloadProgressBar	The DownloadProgressBar is not mirrored. It always fills from left to right.
Image	The contents of the Image control are not mirrored. To mirror the contents of an MX Image control, set the value of the `layoutDirection` property directly on the component. You cannot mirror the contents of the Spark Image control. Use the Spark BitmapImage or MX Image control instead.
Link	Button order is from right to left. All icons are aligned on the right of each button.
NumericStepper	The text in the input is aligned to the right. The navigation buttons are on the left. Numbers appear the same.
PopUpButton	The text is aligned to the right. The drop down icon is on the left.
PopUpMenuButton	See PopUpButton and Menu.
ProgressBar	The ProgressBar control fills in the track from right to left. The label is aligned to the right. The ProgressBar also has a direction style property that you can use to set the direction of the filling.
RadioButton	Labels are on the left of the radio icon.
RadioButtonGroup	See RadioButton control.
ScrollBar, HScrollBar, VScrollBar	Vertical scrollbars are the same, regardless of the value of the `layoutDirection` property. For horizontal scrollbars, the thumb defaults to the right, which is scroll position 0.
Slider, HSlider, VSlider	For the HSlider control, the slider thumb's starting point is reversed. The minimum value is at the right-most point on the slider, and the maximum value is at the left-most point.
Spinner	This component does not change when mirrored.
SWFLoader	The SWFLoader control mirrors its contents. In some cases, this might be undesireable. The workaround is to explicitly set the value of the `layoutDirection` property on the SWFLoader control when you do not want it to mirror the contents.
VideoDisplay	The contents of the VideoDisplay control are not mirrored, unless you explicitly set `layoutDirection` directly on the control.
VideoPlayer	The contents of the VideoPlayer control are never mirrored. The player controls are mirrored so that the slider thumb starts on the right and the location of the play button is on the right.

List-based controls

Control	Description
ButtonBar	Button order is from right to left. All icons are aligned on the right of each button.
ComboBox	In the editable field, text is aligned to the right.
DropDownList	The icon appears on the left side. List items are right aligned.
FileSystemComboBox	Same as the MX ComboBox control.
FileSystemList	Same as the MX List control.
List, HorizontalList	All items in the List are aligned to the right. ScrollBars are on the left (if they are necessary). For a horizontal list, item 0 begins on the right. Items are laid out right to left. A horizontal scroll bar thumb is positioned at `scrollPosition` 0 on the right. Note that for HorizontalList, the `direction` property requires a value of `TileBaseDirection.HORIZONTAL` or `TileBaseDirection.VERTICAL`. The `direction` style supports "ltr" and "rtl". The `direction` property on the MX List and Spark List controls cannot be set in ActionScript.

Containers

Control	Description
Accordion	The labels and icons are aligned to the right in the Accordion headers. All of the child controls of the Accordion panes inherit the `layoutDirection="rtl"` setting, unless the child controls override this value, either in the application or in the defaults.css file.
DividedBox	The first content area of a DividedBox is on the right.
Form	Form item labels are on the right on controls. The form heading is aligned right.
Grid	Grid items move from right to left and top to bottom.
HTML (AIR only)	The contents of the HTML control are not mirrored.
Panel	The title and any icons are aligned right. All of the child controls of the Panel container inherit the `layoutDirection="rtl"` setting, unless the child controls override this value, either in the application or in the defaults.css file.
TabNavigator	Tabs are laid out from right to left.
TileList	The first item is at the top right of the TileList. Items are laid out from right to left. The direction property requires a value of `TileBaseDirection.HORIZONTAL` or `TileBaseDirection.VERTICAL`. The `direction` style supports "ltr" and "rtl". The `direction` property on the TileList control cannot be set in ActionScript.
TitleWindow	The title is aligned on the right. The close button is aligned on the left.
Window	The title is aligned at the top right. All children are right aligned.
WindowedApplication (AIR only)	The WindowedApplication title is aligned at the top right. Controls of the WindowedApplication are at the top left. Status text is aligned on the right.

Data-driven controls

Control	Description
MX chart controls	For CartesianChart-based charting controls, the point of origin is the lower right. To align the DataTips to the right, use the `direction` style property. For more information, see "Mirroring charting controls" on page 2087.
DataGrid	Headers and all text are aligned right. Sort arrows are aligned left. Focus in editors moves from right to left. Column 0 is on the far right.
FileSystemDataGrid	Same as the MX DataGrid control.
FileSystemTree	Same as the MX Tree control.
Menu	Items are aligned right and submenus open to the left.
MenuBar	MenuBar data provider items are laid out from right to left. Submenus open to the left.
Tree	Items are aligned on the right. Items open to the left. The icons are reversed.

Text-based controls

The Spark text-based controls support bidirectionality. They auto-detect the direction of text based on the character codes. MX text-based controls do not support mirroring or bidirectionality. When using Spark text-based controls, you should set the `direction` property on the parent container or application. Setting the `direction` property causes the control to render punctuation properly as well as place the cursor in the correct location when the control gains focus.

For most text controls, changing the layout direction has no effect. This is mostly because most text controls do not have chrome that would require laying out differently. The following table describes the effects of changing `layoutDirection` on text controls:

Control	Description
Label	No change when `layoutDirection="rtl"`.
RichEditableText	No change when `layoutDirection="rtl"`.
RichText	No change when `layoutDirection="rtl"`.
RichTextEditor	No change when `layoutDirection="rtl"`. Note that the RichTextEditor control does not support mirroring or bidirectional text.
Text	No change when `layoutDirection="rtl"`. Note that the Text control does not support mirroring or bidirectional text.
TextArea	Spark TextArea: Text is aligned right. ScrollBars are on the left. Cursor starts on the right of the TextInput control, but only if `direction="rtl"`. MX TextArea: Does not support mirroring or bidirectional text. Use the Spark TextArea control instead.
TextInput	Spark TextInput: Text is aligned right. Cursor starts on the right of the TextInput control, but only if `direction="rtl"`. MX TextInput: Does not support mirroring or bidirectional text. Use the Spark TextInput control instead.

For more information, see "Mirroring with text controls" on page 2077.

Mirroring with text controls

Text in applications that use layout mirroring should generally read from right to left. Chrome on the text controls should lay out with an emphasis on right to left navigation.

The Flex compiler auto-detects the direction that text should read based on the Unicode values of the characters. When the characters use an RTL language such as Arabic or Hebrew, TLF-based text controls reverse the order in which they are written.

Flex adjusts the location of punctuation marks such as periods, exclamation points, and parentheses when the `direction` is set to "rtl". In addition, Flex also adjusts number separators, paragraph breaks, and soft hyphens, as well as other non-character entities so that their position makes sense. The algorithms for this are described in Description of the Unicode bidirectional algorithm.

The direction of the text itself is only reversed if the character set contains codes that instruct the compiler to read from RTL.

All text-based controls in the Spark component set support the `direction` property because they support FTE/TLF. Many text-based controls in the MX component set support the `direction` property.

If your application uses text-based controls in the MX component set such as MX Label, you must configure the compiler to use FTE, which supports bidirectional text. Otherwise, the text will render from LTR regardless of the character set you use. To ensure that MX controls use FTE in Flash Builder, select the "Use Flash Text Engine in MX Components" option. On the command line, you can apply the MXFTEText.css theme file:

```
mxmlc -theme+=themes/MXFTEText.css MyApp.mxml
```

Selecting this option causes most internal dependencies on UITextField to be replaced with UIFTETextField, which supports FTE. The MX TextInput, TextArea, and RichText text classes do not include this support. You should replace these controls with the equivalent Spark text-based controls.

For text controls that have columns, the `direction` style property also affects the column order. When `direction` is set to "rtl", the first column is on the right.

The MX RichTextEditor control does not support mirroring or bidirectional text.

Mirroring with skins

If you create custom skins, you should be aware of the effects on the skins when they are mirrored. In general you do not have to change your skins to use mirroring. In some cases, however, skins use absolute positioning to draw elements of the skin. You might need to edit the skin classes to make them appear correctly when setting `layoutDirection` to "rtl".

Mirroring is supported for components that use the Spark and Wireframe themes. It is not supported for components that use the Halo theme, or when you set the `compatibility-version` compiler option to 3.0.0.

Mirroring with effects

Effects that specify a direction, such as Wipe, or that incorporate a direction, such as WipeRight and WipeLeft, are not reversed when `layoutDirection` is set to "rtl". The effect always wipes in the direction that is defined by the effect, regardless of the value of the `layoutDirection` property.

The following example shows that when you change the `layoutDirection`, the direction of the Wipe does not change:

```
<?xml version="1.0"?>
<!-- mirroring\EffectMirroring.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:states>
        <s:State name="default"/>
        <s:State name="vis"/>
    </s:states>

    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence target="{myB}">
                <s:Wipe id="wipeEffect"
                        direction="right"
                        duration="1000"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>
    <fx:Script>
        <![CDATA[
            private function setDirectionProps():void {
                this.setStyle('layoutDirection', ddl1.selectedItem);
            }
        ]]>
    </fx:Script>

    <s:Panel title="Wipe Effect">
        <s:layout>
            <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
        </s:layout>
        <s:Button id="myB" label="Wipe Effect"
          visible.default="false" visible.vis="true"/>
        <s:Button label="Show Button"
                  click="currentState='vis'"/>
        <s:Button label="Hide Button"
                  click="currentState='default'"/>
    </s:Panel>
    <s:DropDownList id="ddl1" requireSelection="true" selectedIndex="0"
change="setDirectionProps()">
        <s:ArrayList source="['ltr','rtl']"/>
    </s:DropDownList>
</s:Application>
```

Effects that specify x and y coordinates, such as Zoom and Move, apply the transform operations relative to the object. Because mirroring changes the location of the x coordinate, these effects are effectively reversed when the value of the `layoutDirection` property is changed.

The following example shows that when the `layoutDirection` changes, the zoom effect changes its destination point:

```
<?xml version="1.0"?>
<!-- mirroring\MoveEffectMirroring.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <s:Move id="moveOut" target="{targetImg}" xBy="100"/>
        <s:Move id="moveBack" target="{targetImg}" xBy="-100"/>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            private function setDirectionProps():void {
                this.setStyle('layoutDirection', ddl1.selectedItem);
            }
        ]]>
    </fx:Script>
    <s:Panel title="Move Effect Example" height="200" width="300">
        <s:Image id="targetImg" source="@Embed(source='../assets/logosmall.jpg')"/>
        <s:Button id="b1"
                  left="5" bottom="5"
                  label="Move Out" click="moveOut.play();"/>
        <s:Button id="b2"
                  left="120" bottom="5"
                  label="Move Back" click="moveBack.play();"/>
    </s:Panel>
    <s:DropDownList id="ddl1" requireSelection="true" selectedIndex="0"
change="setDirectionProps()">
        <s:ArrayList source="['ltr','rtl']"/>
    </s:DropDownList>
</s:Application>
```

When using coordinate-based effects with controls that do not inherit the `layoutDirection` style property, you might need to customize the target of the effect.

For example, if you apply the Scale effect to an Image control while the application is RTL, the Image's coordinates will not be mirrored. The result is that the Scale effect's point of origin is the upper left even when the application is RTL. To have the effect play correctly, you might need to explicitly set the Image control's `layoutDirection` to "rtl", but then the image itself is reversed as well.

The following example shows how the effect plays incorrectly when the layout for the application is RTL, but the layout for the image is not RTL:

```
<?xml version="1.0"?>
<!-- mirroring\ScaleEffectMirroring.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:Scale id="scaleUp" target="{targetImg}" scaleXBy="2" scaleYBy="2"/>
        <s:Scale id="scaleDown" target="{targetImg}" scaleXBy="-2" scaleYBy="-2"/>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            private function setAppDirection():void {
                this.setStyle('layoutDirection', ddl1.selectedItem);
            }
            private function setImageDirection():void {
                /* Note that this example uses an MX Image control rather than a Spark Image 
control.
                    Spark images cannot be reversed with the layoutDirection property. */
                targetImg.setStyle('layoutDirection', ddl2.selectedItem);
            }
        ]]>
    </fx:Script>
    <s:Panel title="Scale Effect Example" height="300" width="300">
        <s:layout>
            <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
        </s:layout>

        <mx:Image id="targetImg" source="@Embed(source='../assets/logosmall.jpg')"/>
        <s:Button id="b1" label="Increase" click="scaleUp.transformX=-100;scaleUp.play();"/>
        <s:Button id="b2" label="Decrease" click="scaleDown.play();"/>
    </s:Panel>
    <s:HGroup>
        <s:Label text="Set Application's layoutDirection:"/>
        <s:DropDownList id="ddl1" requireSelection="true" selectedIndex="0" 
change="setAppDirection()">
            <s:ArrayList source="['ltr','rtl']"/>
        </s:DropDownList>
    </s:HGroup>
    <s:HGroup>
        <s:Label text="Set Image's layoutDirection:"/>
        <s:DropDownList id="ddl2" requireSelection="true" selectedIndex="0" 
change="setImageDirection()">
            <s:ArrayList source="['ltr','rtl']"/>
        </s:DropDownList>
    </s:HGroup>
</s:Application>
```

To workaround this issue, you should use two separate images, one for when the application’s layout is RTL and one for when the application’s layout is LTR. Then load the appropriate image when the layout changes.

Mirroring graphics and video

Images and video controls

Image and video controls do not inherit layout direction. The BitmapImage, Spark and MX Image, VideoPlayer, and VideoDisplay controls do not mirror their contents when you set the `layoutDirection` property on the application or parent container. These controls override the value of this property in the globals.css file because the expected behavior of images or videos is that the contents are not mirrored.

You can override this behavior in most cases, so that the contents of these controls is mirrored. You do this by setting the value of the `layoutDirection` property to “rtl” directly on the control. This is true for all the controls mentioned except Spark Image. The Spark Image control does not mirror its contents regardless of the value of the `layoutDirection` property because it is actually a child of the BitmapImage control (and image-related controls do not inherit the layout direction). To mirror the image, use the Spark BitmapImage or MX Image control.

As with images, icons are typically not mirrored, even if the component that uses the icon is reversed (except in some special cases, such as the disclosure icon in a Tree control).

To mirror the contents of the MX Image, BitmapImage, or VideoDisplay controls, set the `layoutDirection` property directly on the control. You cannot mirror the contents of the VideoPlayer control or the Spark Image control.

The following example sets the `layoutDirection` on the Application tag to "rtl", and then sets the `layoutDirection` property to "rtl" directly on several of the images. This example shows that the first image is not mirrored because it does not override the `layoutDirection` property and does not inherit its value from the Application container. The second image is also not mirrored because the Spark Image control does not reverse when the property is set on it. The third and fourth images are mirrored because the MX Image and Spark BitmapImage controls support setting `layoutDirection` directly on them.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring\MirroredImage.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
       xmlns:s="library://ns.adobe.com/flex/spark"
       xmlns:mx="library://ns.adobe.com/flex/mx"
       width="750"
       layoutDirection="rtl">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Label text="application.layoutDirection = rtl"/>

    <s:HGroup>
        <s:Panel title="Spark Image with default layoutDirection" height="150" width="250">
            <s:layout>
                <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
            </s:layout>
            <s:Image source="@Embed(source='../assets/logosmall.jpg')"/>
        </s:Panel>

        <s:Panel title="Spark Image with layoutDirection=rtl" height="150" width="250">
            <s:layout>
                <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
            </s:layout>
```

```
      <s:Image source="@Embed(source='../assets/logosmall.jpg')" layoutDirection="rtl"/>
        </s:Panel>
    </s:HGroup>
    <s:HGroup>
        <s:Panel title="MX Image with layoutDirection=rtl" height="150" width="250">
            <s:layout>
                <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
            </s:layout>
     <mx:Image source="@Embed(source='../assets/logosmall.jpg')" layoutDirection="rtl"/>
        </s:Panel>

        <s:Panel title="BitmapImage with layoutDirection=rtl" height="150" width="250">
            <s:layout>
                <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
            </s:layout>
            <s:BitmapImage source="@Embed(source='../assets/logosmall.jpg')"
layoutDirection="rtl"/>
        </s:Panel>
    </s:HGroup>
</s:Application>
```

You can also cause the contents of image controls to be mirrored by setting the `scaleX` property to -1 rather than setting the `layoutDirection` property; for example:

```
sparkImage.scaleX = -1;
```

FXG and MXML graphics

FXG graphics and MXML graphics are mirrorred by default. When the application or parent container sets the `layoutDirection` property, the FXG component and the MXML graphic are mirrorred.

The following example shows a simple FXG component, and an MXML graphic, that inherit the application's `layoutDirection`, and are therefore mirrored:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring\MirroredFXG.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
       xmlns:s="library://ns.adobe.com/flex/spark"
       xmlns:mx="library://ns.adobe.com/flex/mx"
       xmlns:custom="*"
       layoutDirection="rtl"
       width="600" height="300">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Label text="application.layoutDirection = rtl"/>
    <s:HGroup>
        <s:Panel title="FXG: layoutDirection = default" height="150" width="250">
            <s:layout>
                <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
            </s:layout>
            <custom:ArrowAbsolute/>
        </s:Panel>
        <s:Panel title="MXML: graphic layoutDirection = default" height="150" width="250">
            <s:layout>
                <s:VerticalLayout paddingTop="15" paddingLeft="15"/>
            </s:layout>
            <!-- Use absolute coordinates. -->
            <s:Graphic>
                <!-- Use Use compact syntax with absolute coordinates. -->
                <s:Path data="
                        M 20 0
                        C 50 0 50 35 20 35
                        L 15 35
                        L 15 45
                        L 0 32
                        L 15 19
                        L 15 29
                        L 20 29
                        C 44 29 44 6 20 6">
                    <!-- Define the border color of the arrow. -->
                    <s:stroke>
                        <s:SolidColorStroke color="0x888888"/>
                    </s:stroke>
                    <!-- Define the fill for the arrow. -->
                    <s:fill>
                        <s:LinearGradient rotation="90">
                            <s:GradientEntry color="0x000000" alpha="0.8"/>
                            <s:GradientEntry color="0xFFFFFF" alpha="0.8"/>
                        </s:LinearGradient>
                    </s:fill>
                </s:Path>
            </s:Graphic>
        </s:Panel>
    </s:HGroup>
</s:Application>
```

This example uses the following FXG file as a custom component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/ArrowAbsolute.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <!-- Use Use compact syntax with absolute coordinates. -->
    <Path data="
         M 20 0
         C 50 0 50 35 20 35
         L 15 35
         L 15 45
         L 0 32
         L 15 19
         L 15 29
         L 20 29
         C 44 29 44 6 20 6">
         <!-- Define the border color of the arrow. -->
         <stroke>
              <SolidColorStroke color="#888888"/>
         </stroke>
         <!-- Define the fill for the arrow. -->
         <fill>
              <LinearGradient rotation="90">
                   <GradientEntry color="#000000" alpha="0.8"/>
                   <GradientEntry color="#FFFFFF" alpha="0.8"/>
              </LinearGradient>
         </fill>
    </Path>
</Graphic>
```

You can set the `layoutDirection` of the parent container on the Graphic tag directly; for example:

```
<s:Graphic layoutDirection='rtl'>
```

Flex mirrors the contents of the BitmapFill class when you set the property on the container. This can cause undesireable results when you set the `layoutDirection` property on an application that supports both RTL and LTR. To workaround this issue, you can create two versions of the graphic. Then create a custom skin for your control that uses the background image and use states to pick the right one depending on the value of the application's `layoutDirection`.

Coordinate systems

When using graphics, you specify the start point and end point of paths. These locations are represented as coordinate X/Y pairs such as (0,0). The default location for (0,0) is the upper left corner of the container that you are drawing in. If you change the layout direction, the coordinates, too, are mirrored. As a result, the coordinates (0,0) in a container with `layoutDirection="rtl"` refers to the upper right corner of the container, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring/CoordinatesMirroring.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import mx.core.LayoutDirection;
        ]]>
    </fx:Script>
     <mx:Panel title="LTR: from (0,0) to (100,100)"
        height="75%" width="40%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
           <s:Line xFrom="0" xTo="100" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="1"/>
                 </s:stroke>
           </s:Line>
        </s:Group>
     </mx:Panel>
     <mx:Panel title="RTL: from (0,0) to (100,100)" layoutDirection="{LayoutDirection.RTL}"
        height="75%" width="40%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">
        <s:Group>
           <s:Line xFrom="0" xTo="100" yFrom="0" yTo="100">
                 <s:stroke>
                      <s:SolidColorStroke color="0x000000" weight="1"/>
                 </s:stroke>
           </s:Line>
        </s:Group>
     </mx:Panel>
</s:Application>
```

This same logic applies to layout coordinates. When you set the x and y position of a component in an LTR panel, the coordinates start at the upper left. If the panel's layout direction is RTL, the coordinates start in the upper right. However, calling the `Menu.show(0,0)` method will not pop up a menu at the top right, but rather, the top left because global coordinates are used.

If you use the `localToGlobal()` and `globalToLocal()` methods, or the DisplayObject `transform` property, you should be aware of the difference between a component's `layoutMatrix` and `computedMatrix`. The `layoutMatrix` is used to calculate the component's layout relative to its siblings. The `computedMatrix` incorporates the `layoutMatrix` and an IVisualElement's `postLayoutTransformOffsets`, which include the mirroring transform. The `localToGlobal()` and `globalToLocal()` methods and the `transform.concatenatedMatrix` property reflect the computed matrix.

To transform a point to global coordinates using only the `layoutMatrix`, use the `MatrixUtil.getConcatenatedMatrix()` method, as the following example shows:

```
var myGlobalPoint =
MatrixUtil.getConcatenatedComputedMatrix(myIVisualElement).transformPoint(myPoint);
```

When working with coordinates, you should be aware of how the mirroring transform works. In an LTR application, X increases to the right and the origin is the upper left corner. In an RTL layout, X increases to the left and the origin is the upper right corner. This is accomplished internally by scaling the X-axis by -1 and translating the UIComponent by its width, after layout.

The effect of this mirroring transform is as follows:

```
element.x += width;
element.scaleX *= -1;
```

All of the IVisualElement classes implement the mirroring transform using the same mechanism that is used by the offsets transform. That means that they are combined with the layout transform after layout has occurred. The transforms do not affect the actual values of the UIComponent's `x` and `scaleX` properties.

Mirroring DataGrid controls

You can use mirroring with a DataGrid control by setting the `layoutDirection` property on the application. Like most controls, the DataGrid control inherits this value. Setting the layout direction to RTL causes the columns to be arranged from right to left (column 0 is the right-most). This also aligns the sort arrows to the left of the header text. If the DataGrid is editable and the user clicks on a cell, the focus starts on the right side of the grid.

If you use an RTL character set such as Hebrew or Arabic in the cells of a DataGrid control, you must set the "Use Flash Text Engine in MX components" check box in Flash Builder. Otherwise, the characters will not render correctly because the TextField control does not support bidirectionality.

To align the text in the cells to the right, reverse the direction of the header text, and reverse the text in the individual cells of a DataGrid, you must also set the `direction` style property to "rtl". You can set this on a parent container or on the DataGrid itself.

The following example sets both the `layoutDirection` and `direction` properties to "rtl" so that the entire Spark DataGrid is mirrored:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring\MirroredDataGrid.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
   xmlns:s="library://ns.adobe.com/flex/spark"
   xmlns:mx="library://ns.adobe.com/flex/mx"
   width="600" height="400"
   layoutDirection="rtl" direction="rtl">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var dp:ArrayCollection = new ArrayCollection([
                {Artist:'Train', Album:'Drops of Jupiter', Price:13.99},
                {Artist:'Charred Walls of the Damned', Album:'Ghost Town', Price:8.99},
                {Artist:'Bleading Deacons', Album:'Rule the Night', Price:11.99},
                {Artist:'Three Stooges', Album:'Greatest Hits', Price:9.99} ]);
        ]]>
    </fx:Script>

    <s:DataGrid id="myGrid" dataProvider="{dp}"
        width="350" height="150"/>
</s:Application>
```

Mirroring layouts

In a mirrored container, the roles of the left and right layout constraints are effectively reversed: the left constraint specifies the distance from the component's right edge to the right edge of the container, and the right constraint specifies the distance between the left edges.

Mirroring charting controls

To mirror charts, you set the `layoutDirection` property on the chart's parent container to RTL. Typically, you set it on the Application container. Charting controls inherit the value of this property set on the parent container.

In charts that are based on the CartesianChart class, such as the AreaChart, LineChart, and PlotChart controls, setting the `layoutDirection` property to "rtl" sets the point of origin at the lower right of the chart, rather than the lower left. For charts based on the PolarChart class, such as the PieChart control, setting the `layoutDirection` property to "rtl" has no effect on the point of origin.

The following example shows a chart that uses mirroring:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring\MirroredChart.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
       xmlns:s="library://ns.adobe.com/flex/spark"
       xmlns:mx="library://ns.adobe.com/flex/mx"
       creationComplete="srv.send()"
       layoutDirection="rtl"
       width="600" height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:HTTPService id="srv" url="http://aspexamples.adobe.com/chart_examples/expenses-
xml.aspx"/>
    </fx:Declarations>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        @namespace charts "mx.charts.chartClasses.*";

        charts|DataTip {
            direction:rtl;
            paddingRight:-10;
            paddingLeft:5;
        }
    </fx:Style>

    <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart" dataProvider="{srv.lastResult.data.result}"
showDataTips="true">
```

```
            <mx:horizontalAxis>
                <mx:CategoryAxis id="haxis" categoryField="month" title="Month"/>
            </mx:horizontalAxis>
            <mx:verticalAxis>
                <mx:LinearAxis id="vaxis" title="Amount"/>
            </mx:verticalAxis>

            <mx:series>
                <mx:ColumnSeries
                    xField="month"
                    yField="profit"
                    displayName="Profit"/>
                <mx:ColumnSeries
                    xField="month"
                    yField="expenses"
                    displayName="Expenses"/>
            </mx:series>
        </mx:ColumnChart>
    </s:Panel>
</s:Application>
```

If you use an RTL character set such as Hebrew or Arabic in the text-based components of a chart control (such as the axis labels, axis titles, and DataTips), you must set the "Use Flash Text Engine in MX components" check box in Flash Builder. Otherwise, the text will render from left to right, regardless of the character set. In addition, you should be aware of some minor differences in the way charting controls use spacing and alignment when the `layoutDirection` is "rtl". For example, to right align the contents of DataTips, set the DataTip `direction` style property to "rtl". You might also need to adjust the padding properties to get the alignment you want. For example:

```
@namespace charts "mx.charts.chartClasses.*";
charts|DataTip {
    direction:rtl;
    paddingRight:-10;
    paddingLeft:5;
}
```

In addition, when the `layoutDirection` property is set to "rtl":

- Gutter padding — The left and right gutter style properties are reversed. The `gutterLeft` property changes the gutter padding on the right, and the `gutterRight` property changes the gutter padding on the left.
- Axis labels — Setting `labelAlign="left"` aligns the label to the right, and `labelAlign="right"` aligns the label to the left.
- Axis titles — The `textIndent` property is unaffected by the `layoutDirection` property. Axis titles do not support the `textAlign` property, so setting the `layoutDirection` property has no effect on this property.

Using the LocaleID class

The LocaleID class, available in Flash Player 10.1, provides methods and properties for using locale IDs. Locale IDs are strings that represent a particular locale. They can be useful when deciding whether to change the layout direction of an application.

The LocaleID class includes the `isRightToLeft()` method, which returns `true` if the currently set locale uses an RTL character code set, but otherwise returns `false`. You can use this method to set the values of the `layoutDirection` and `direction` properties. When you change locales in the following example, the application checks the `isRightToLeft()` method and sets the `layoutDirection` and `direction` properties on the Application object accordingly:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mirroring\LocaleIDMirroringExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
            xmlns:s="library://ns.adobe.com/flex/spark"
            xmlns:mx="library://ns.adobe.com/flex/mx"
            width="300" height="300"
            layoutDirection="{localeBasedDirection}"
            direction="{localeBasedDirection}"
            creationComplete="updateSelection()">
    <fx:Script>
        <![CDATA[
            import flash.globalization.LocaleID;
            import mx.collections.ArrayCollection;
            [Bindable]
            public var localeBasedDirection:String;
            [Bindable]
            public var locID:LocaleID;
            [Bindable]
            public var myDP:ArrayCollection = new ArrayCollection(
                [ {name:"US", locale:"en_US"},
                    {name:"Israel", locale:"he_IL"},
                    {name:"France", locale:"fr_FR"} ]);

            private function updateSelection():void {
                locID = new LocaleID(localeIdList.selectedItem.locale);
                isRTLLabel.text = "RTL: " + locID.isRightToLeft().toString();
                getRegionLabel.text = "Region: " + locID.getRegion();
                getLanguageLabel.text = "Language: " + locID.getLanguage();

                if (locID.isRightToLeft()) {
                    localeBasedDirection = "rtl";
                } else {
                    localeBasedDirection = "ltr";
                }
            }
        ]]>
    </fx:Script>
    <s:Panel title="Select Locale" height="200" width="250">
        <s:layout>
            <s:VerticalLayout paddingLeft="10" paddingRight="10" paddingTop="10"/>
        </s:layout>
        <s:VGroup>
            <s:DropDownList id="localeIdList" selectedIndex="0"
                dataProvider="{myDP}"
                labelField="name"
                change="updateSelection();"/>
            <s:Label id="isRTLLabel"/>
            <s:Label id="getRegionLabel"/>
            <s:Label id="getLanguageLabel"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

This example requires classes in the flash.globalization.* package, which are available in the 10.1 or later playerglobal.swc file. When compiling this example, be sure to set the target player version to 10.1 or later. In Flash Builder, this option is on the Compiler properties page. On the command-line, you can set the target Player version by using the `target-player` option; for example:

```
mxmlc -target-player=10.1 MyApp.mxml
```

In your applications, you will typically have some additional logic that sets the locale in your application, such as looking at the OS language code or accepting a user-configured setting. For information on setting locales in your application, see “Setting the locale” on page 2056. For more information about using the classes in the flash.globalization.* package, see Flash Player: Cultural diversity without complexity.

Mirroring with ToolTips and Error tips

The `layoutDirection` of the ToolTip class is same as the component that is attached to the ToolTip, or LTR, if the component does not implement the ILayoutDirectionElement interface.

Overlapping direction properties

Some components have a `direction` style property that is not related to text direction. These `direction` style properties typically take values not related to text direction. The compiler could throw an error if you try to set the `direction` property with these values. The following table lists these components:

Control	**Description**
mx.containers.Box	The `direction` property refers to the direction in which the container lays out its children.
mx.containers.FormItem	The `direction` property detemines whether the children of the FormItem are stacked vertically or placed horizontally.
mx.controls.ProgressBar	The `direction` property refers to the direction in which the bar expands towards completion. Possible values are "left" and "right". In general, you should set this to "right" when the application's layout is RTL.
mx.controls.scrollClasses.ScrollBar	The `direction` property detemines whether the scrollbar is for scrolling vertically or horizontally.
mx.controls.sliderClasses.Slider	The `direction` property detemines whether the slider is oriented vertically or horizontally. If you use an HSlider or VSlider, the control sets the value for you.
mx.containers.Tile	The `direction` property refers to the direction in which the container lays out its children.
mx.controls.listClasses.TileBase (which includes TileList and HorizontalList)	The `direction` property refers to the direction in which the container lays out its children.

Tips for using mirroring

Use the following tips when creating applications that use layout mirroring:

- Avoid setting the `layoutDirection` and direction properties on individual components. Set them on the Application container.
- Avoid using the alignment properties of individual components or blocks of text.
- Avoid using absolute positioning.

- If you want to mirror the contents of images (including icons) and videos, you should reverse the image or video in your image editing tools prior to importing them into your Flex application. The Spark and MX Image, BitmapImage, and VideoPlayer controls do not inherit the `layoutDirection` property from their parent container. The MX Image, Spark BitmapImage, and VideoDisplay controls do mirror their contents if you set `layoutDirection` on the control directly. The Spark Image control does not mirror its contents regardless of the value of its layout direction.
- Spark text controls do not use layout mirroring. Instead, they support bidirectionality of text internally. As a result, the primitive text controls such as Label, RichText, and RichEditableText do not support the `layoutDirection` property. You typically set the `direction` style property to change the directionality of the text in these controls.
- If you use text-based controls from the MX component set in your application, be sure to configure the compiler to use FTE to display the text. To do this in Flash Builder, select the "Use Flash Text Engine in MX components" check box. To do this on the command line, apply the MXFTEText.css theme file.
- For Spark text controls or FTE text in MX components, use the values `TextAlign.START` and `TextAlign.END` for the `textAlign` property, rather than `TextAlign.LEFT` and `TextAlign.RIGHT`. Using `START` and `END` will work in the expected manner, regardless of the direction of the text.
- In many components, navigating with the arrow keys changes depending on the directionality of the text. For example, the left arrow will navigate toward the beginning of the text and the right arrow key will navigate toward the end when the text is RTL.

Resource Bundles

Adobe® Flex® provides support for resource bundles in your applications.

Introduction to resource bundles

A *resource bundle* is a list of assets that is organized by locale. These assets can be just about anything, from strings, numbers, formats, and images to styles. You typically have a separate resource bundle for each locale that your application supports.

To localize an application with resource bundles, you first create properties files that define the localized assets. You then compile these properties files into the application as resource bundles, or create resource modules from the properties files and load them at run time.

Flex lets you change locales on the fly; if you compile more than one resource bundle into an application, you can toggle the resource bundle based on the locale. In addition, if you compile resource modules, you can load and unload the SWF files at run time based on the locale. You can even create new resource bundles programmatically.

How you load your resource bundles depends on how many locales your application supports:

- If your application supports just one or two locales, then you typically compile all the resources into the application. For more information, see “Using resource bundles” on page 2096.
- If your application supports many locales, you will likely want to load the appropriate resources at run time rather than compile all supported resources into the application at compile time. To do this, you compile your resource bundles into resource modules. For more information, see “Using resource modules” on page 2105.

If your application only needs to format numbers, currencies, and sort orders, you can use the built-in support for localization in the spark.globalization.* classes without using resource bundles.

More Help topics

"Localization" on page 2055

Lupo Manager, an AIR application that manages resource bundles

Creating resources

You define resource bundles in *resource properties files*. These properties files contain key/value pairs and are in UTF-8 format. You commonly use these properties files to specify the values of Strings in your applications, such as the label on a button or the items in a drop down list. The following example specifies the values for a form's labels in English:

```
# locale/en_US/RegistrationForm.properties
registration_title=Registration
submit_button=Submit Form
personname=Name
street_address=Street Address
city=City
state=State
zip=ZIP Code
thanks=Thank you for registering!
```

Resources can also embed binary assets such as audio files, video files, SWF files, and images. To embeds these assets in your properties files, you use the `Embed()` directive, just as you would include assets in a runtime style sheet. The following example embeds a JPG file as a resource:

```
flag=Embed("images/unitedstates.jpg")
```

You can also extract symbols from an embedded SWF file when using the `Embed()` directive, as the following example shows:

```
flag=Embed(source="FlagsOfTheWorld.swf", symbol="unitedstates")
```

To include custom classes, such as a programmatic skin, you can use the `ClassReference()` directive. The following example embeds the MySorter class in the sortingClasses.en_US package:

```
SORTER=ClassReference("sortingClasses.en_US.MySorter")
```

For information on how to use these various types of resources in your application, see "Using resource bundles" on page 2096.

You typically store resource properties files in a locale/*locale_name* subdirectory. You add this directory to your source path so that the compiler can find it when you compile your application, and append the locale to the `locale` compiler option. For information on how to compile resources into your Flex application, see "Compiling resources into Flex applications" on page 2092.

Compiling resources into Flex applications

After you create the resource properties files, you can either compile them as resource bundles into your application or you compile them into resource modules and load them at run time. If you compile the resources into the application, the compiler converts them to subclasses of the ResourceBundle class, and adds them to the application at compile time. If you compile the resources into resource modules, the compiler converts them to SWF files that you then load at run time on an as-needed basis.

For information on compiling and using resource modules, see "Using resource modules" on page 2105.

Compile resources into the application on the command line

1 Specify one or more locales to compile the application for with the `locale` compiler option. The value of this option is used by the `source-path` option to find the resource properties files. The `library-path` option also uses this value to include localized framework resources.

2 Specify the location of the resources with the `source-path` option. You can add the resources for more than one locale by using the `{locale}` token.

3 Set the value of the `allow-source-path-overlap` compiler option to `true`. This is optional, but if you do not set it, you might get a warning that the source path for the locale is a subdirectory of the project's source path.

In the following example, the LocalizedForm.mxml application uses a custom resource bundle for a single locale, en_US:

```
mxmlc -locale=en_US -source-path=c:\myapp\locale\{locale} -allow-source-path-overlap=true
c:\myapp\LocalizedForm.mxml
```

In Adobe® Flash® Builder™, you add the resource directories to the project's source path, and then add additional compiler options. The `source-path` option tells the compiler where to look for additional source files such as properties files.

Add resource directories to the project's source path in Flash Builder

1 Select Project > Properties.

2 Select the Source Path tab.

3 Click the Add Folder button. The Add Folder dialog box appears.

4 Navigate to the resource properties files' parent directory. Typically, you have locales in parallel directories under a parent directory such as /loc or /locale. For the en_US locale, you might have c:/myapp/loc/en_US directory. You will want to generalize the source path entry so that all locales are included. In this case, append `{locale}` to the end of the directory path. For example:

   ```
   c:/myapp/loc/{locale}
   ```

   This instructs the compiler to include all directories that match the `{locale}` token. In this case, en_US. If you add other locales, you only need to add them to the `locale` option and not the source path, as long as the new locale's resources are parallel to the existing locale's resources. For more information, see "Adding new locales" on page 2095.

5 Click OK to add this directory to your project's source path.

You then add the `locale` and `allow-source-path-overlap` options to the Additional Compiler Arguments field on the Flex Compiler pane in the project's properties, as the following example shows:

```
-locale=en_US -allow-source-path-overlap=true
```

The `locale` option instructs the compiler which resource properties files to convert into resource bundles. By default, all locales listed under the framework/bundles directory are supported (including en_US, de_DE, ru_RU, and ja_JP). If you want to use resource bundles for other locales, you must also generate the framework resource bundles for those locales and add those locale's to the `locale` option. For more information, see "Adding new locales" on page 2095.

The value of the `locale` option (in this case, en_US) is also used by the `library-path` option. If you specify multiple locales or change the locale, you will not be required to also change the library path.

The `allow-source-path-overlap` option lets you avoid a warning that the MXML compiler generates. You typically create a new subdirectory named locale under your project and then a subdirectory under that for each locale. You must explicitly add the locale directory to your source path. The directory where the application's MXML file is located is implicitly added to the source path. In the default Flash Builder project layout, this folder is the project directory and it is therefore the parent of the locale directory. By settings the allow-source-path-overlap option to `true`, you instruct the compiler not to warn you about overlapping values in the source path.

You can specify the locales in the flex-config.xml file by using the `<locale>` tag, as the following example shows:

```
<locale>
    <localeElement>en_US</localeElement>
</locale>
```

To add entries to your source path in the configuration file, you use the `<source-path>` child tag, as the following example shows:

```
<source-path>
    <path-element>locale/{locale}</path-element>
</source-path>
```

The `<source-path>` tag is commented out by default, so you must first uncomment the tag block.

Properties file syntax

Properties files are parsed as they are in Java. Each line typically takes the form of key=value. The following rules apply to properties files:

- Lines in properties files are not terminated by semi-colons or other characters.
- You can use an equals sign, a colon, or whitespace to separate the key from the value; for example:

  ```
  key = value
  key : value
  key value
  ```

- To add a comment to your properties file, start the line with a # or !. You can insert whitespace before the # or ! on a comment line. The following are examples of comments in a properties file:

  ```
  ! This is a comment.
  # This is a comment.
  ```

- Whitespace at the beginning of a value is stripped. Trailing whitespace is not stripped from the value.
- You can use standard escape sequences such as \n (newline), \r (return), \t (tab), \u0020 (space), and \\ (backslash).
- Backslash-space is an escape sequence for a space; for example, if a value starts with a space, you must write it as backslash-space or the compiler will interpret it as optional whitespace preceding the value. You are not required to escape spaces within a value. The following example starts the value with a space:

  ```
  key =\ value
  ```

- You can continue a line by ending it with a backslash. Leading whitespace on the next line is stripped.
- Backslashes that are not part of an escape sequence are removed. For example, \A is just A.
- You are not required to escape a double quote or a single quote.
- Lines that contain only whitespace are ignored.

Adding new locales

To add new locales to your projects:

1. Add the locale to the `locale` compiler option. This is typically a comma-separated list, as the following example shows:

    ```
    -locale=en_US,es_ES
    ```

    In Flash Builder, you add this in the Additional Compiler Arguments field of the Flex Compiler properties panel.

    If you edit the flex-config.xml file, you add locales by using the following syntax:

    ```
    <locale>
        <locale-element>en_US</locale-element>
        <locale-element>es_ES</locale-element>
    </locale>
    ```

2. Ensure that the new locale's resource properties files are in the source path. The source path entry for localized resources typically uses the `{locale}` token so that all new locales are automatically added to the source path, as long as those locales' resource directories have the same parent directory.

3. Create the framework resource bundles for the new locale, if the framework resource bundles do not already exist. You can see a list of the supported bundles by looking at the directory list under framework/bundles. The supported list includes en_US, ja_JP, fr_FR, ru_RU, and es_ES.

The `locale` option defines what locale's resources to include on the source path. It also instructs the compiler to include the localized framework resources for the specified locales. Framework components such as Label and Button use resources, just like your application and custom components can use resources. The resources required by these classes are located in the libraries like framework.swc or in separate resource bundle libraries. By default, framework resources for many common locales are included in the Flex SDK.

To use any supported locale, you do not need to do anything other than create properties files for your locale. For locales that are not included, such as en_IN, in addition to creating the new properties files, you must also create new framework locale files before compiling your Flex application.

To create a locale's framework resources, use the copylocale utility in the /sdk/bin directory. For Flash Builder, the copylocale utility is located in *flash_builder_install*/sdks/4.6.0/bin. You can only execute this utility from the command line.

The syntax for the copylocale utility is as follows:

```
copylocale original_locale  new_locale
```

For example, to create the framework locale files for the en_IN locale, use the following command:

```
copylocale en_US en_IN
```

This utility creates a new directory under frameworks/locale/*locale_name*. The name of this directory matches the name of the new locale. This directory is parallel to the other locale directories. In this example, it creates the locale/en_IN directory. This utility also creates SWC files in the new directory, including the framework_rb.swc and rpc_rb.swc files.

These resources must be in the library path. By default, the `locale/{locale}` entry is already set for your `library-path` compiler option, so you do not need to change any configuration options.

When you compile your application, and add the new locale to the `locale` option, the compiler includes the localized framework resources in the SWC files for that new locale.

About the ResourceBundle class

For each resource properties file, the Flex compiler generates a class that extends the ResourceBundle class, and adds that class to the application. The name of the class is the name of the properties file, with an underscore replacing the period in the filename.

The generated ResourceBundle class overrides a single method, `getContent()`, which returns an object that defines the values in the properties file. For example, if you use the RegistrationForm.properties file, the compiler generates the following class:

```
public class RegistrationForm_properties extends ResourceBundle {
    override protected function getContent():Object {
        var properties:Object = {};
        properties["registration_button"] = "Registration";
        properties["submit_button"] = "Submit Form";
        properties["personname"] = "Name";
        properties["street"] = "Street Address";
        properties["city"] = "City";
        properties["state"] = "State";
        properties["zip"] = "ZIP Code";
        properties["thanks"] = "Thank you for registering!";
        return properties;
    }
}
```

To view the generated classes, you can set the `keep-generated-actionscript` compiler option to `true` when you compile your application with an existing resource properties file.

You can write your own classes that extend the ResourceBundle class rather than create resource properties files that the compiler then uses to generate the resource bundle classes. You can then use those classes if you include them in your project. For more information, see "Creating resource bundles at run time" on page 2112.

Using resource bundles

After you create a resource properties file, you can use it in your Flex application as a resource bundle (the compiler converts properties files to subclasses of the ResourceBundle class when you compile your application). You can either bind values from the resource to an expression, or you can use methods of the ResourceBundle class to access those values.

There are two ways to access the values in resource bundles:

- `@Resource` directive
- Methods of the ResourceManager class

Using the `@Resource` directive to include resource bundles in your application is the simplest method. In the directive, you specify the name of the properties file and the key that you want to use. This method is simple to use, but is also restrictive in how it can be used. For example, you can only use the `@Resource` directive in MXML and you cannot change locales at run time. In addition, this directive only returns Strings.

The other way to access resource bundles is through the ResourceManager class. You can use the methods of the ResourceManager class in ActionScript, whereas you can only use the `@Resource` directive in MXML. These methods can return data types other than Strings, such as ints, Booleans, and Numbers. You can also use this class to switch locales at run time, so if you compiled multiple locales into the same application, you can change from one locale to the other.

The following sections describe how to use the `@Resource` directive and the ResourceManager class to access resource bundles in your Flex application.

Using the @Resource directive

In MXML, you can use the `@Resource` directive to access your resource bundles. You pass the `@Resource` directive the name of the resource bundle (the name of the properties file without the .properties extension) and the name of the key from the key/value pairs in the properties file. The directive returns a String of the key's value from your resource bundle.

The following example creates a form; the values for the labels in the form are extracted from the RegistrationForm.properties resource properties file.

```
<?xml version="1.0"?>
<!-- resourcebundles/LocalizedForm.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Form>
        <s:FormItem label="@Resource(key='personname', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='street_address', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='city', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='state', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='zip', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Because this application only uses a single locale, you compile it with the following compiler options:

```
-locale=en_US -allow-source-path-overlap=true -source-path=locale/{locale}
```

Accessing the values in a resource bundle with the `@Resource` directive is restrictive in that the directive can only be used in an MXML tag. In addition, you cannot change the locale at run time.

To use a class, such as a programmatic skin, as a resource, you use the `ClassReference()` directive in your properties file. The following example embeds the MyCheckBoxIcon_en_US class in the properties file:

```
CHECKBOXSKIN=ClassReference("MyCheckBoxIcon_en_US")
```

You then reference that class in your style properties, as the following example shows:

```
<mx:CheckBox selected="true"
    selectedUpIcon="@Resource(key='bundle1', 'CHECKBOXSKIN')"
    selectedDownIcon="@Resource(key='bundle1', 'CHECKBOXSKIN')"
    selectedOverIcon="@Resource(key='bundle1', 'CHECKBOXSKIN')"/>
```

To use a binary asset such as an image, you embed the image in the resource properties file. You can then use the asset anywhere that you might use an embedded image. To embed an image in the properties file, you use the `Embed` directive in your properties file, as the following example shows:

```
flag=Embed("images/unitedstates.gif")
```

The location of the image is relative to the location of the properties file. In this case, the images directory is under locale/en_US.

In your application, you use the `@Resource` directive anywhere a String might supply the location of the image. The following example uses the image as a source for the `<s:Image>` tag:

```
<?xml version="1.0"?>
<!-- resourcebundles/LocalizedFormResourceWithImage.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
        <s:VerticalLayout/>
    </s:layout>
   <s:Image source="@Resource(key='flag', bundle='RegistrationForm')"/>
    <s:Form>
        <s:FormItem label="@Resource(key='personname', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='street_address', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='city', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='state', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='zip', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Using the ResourceManager

To use resource bundles in ActionScript, you use the methods of the ResourceManager class, such as `getString()` and `getClass()`. Using the ResourceManager is more flexible than using the `@Resource` directive because it lets you dynamically rather than declaratively set the values of properties from resources. It also lets you change locales at run time so that all localized resources in your application can be updated at once.

When using the ResourceManager, you must specify metadata that defines the resource bundles for your application. The syntax for this metadata is as follows:

```
<fx:Metadata>
    [ResourceBundle("Resource_file_name")]
</fx:Metadata>
```

For example:

```
<fx:Metadata>
    [ResourceBundle("RegistrationForm")]
</fx:Metadata>
```

For multiple resource bundles, add each one on a separate line inside the same `<fx:Metadata>` tag, as the following example shows:

```
<fx:Metadata>
    [ResourceBundle("RegistrationForm")]
    [ResourceBundle("StyleProperties")]
    [ResourceBundle("FormatterProperties")]
</fx:Metadata>
```

In an ActionScript class, you apply the metadata above the class name, as the following example shows:

```
[ResourceBundle("RegistrationForm")]
public class MyComponent extends UIComponent {
    ...
}
```

You can also bind expressions to the ResourceManager. These expressions are updated any time the locale changes.

The following example uses ActionScript to set the value of the Alert message, and binds the labels in the form to resources by using the ResourceManager’s `getString()` method.

```
<?xml version="1.0"?>
<!-- resourcebundles/LocalizedFormWithBinding.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        import mx.controls.Alert;
        private function registrationComplete():void {
            Alert.show(resourceManager.getString('RegistrationForm', 'thanks'));
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Metadata>
        [ResourceBundle("RegistrationForm")]
    </fx:Metadata>
    <s:Form>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','personname')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','street_address')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','city')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','state')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','zip')}">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
    <s:Button id="b1"
        label="{resourceManager.getString('RegistrationForm','submit_button')}"
        click="registrationComplete()"/>
</s:Application>
```

To use a binary asset such as an image with the ResourceManager, you embed the image in the resource properties file. You can then use the asset anywhere that you might use an embedded image. To embed an image in the properties file, you use the `Embed` directive in your properties file, as the following example shows:

```
flag=Embed("images/unitedstates.gif")
```

The location of the image is relative to the location of the properties file. In this case, the images directory is under locale/en_US.

To use the image in your application, you use the ResourceManager's `getClass()` method. The following example uses the GIF file as both a skin for the Button control and a source for the `<s:Image>` tag:

```
<?xml version="1.0"?>
<!-- resourcebundles/LocalizedFormBindingWithImages.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        import mx.controls.Alert;
        private function initApp():void {
       b1.setStyle("downSkin", resourceManager.getClass("RegistrationForm", "flag"));
        }
        private function registrationComplete():void {
            Alert.show(resourceManager.getString('RegistrationForm', 'thanks'));
        }
    ]]></fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Metadata>
        [ResourceBundle("RegistrationForm")]
    </fx:Metadata>
    <s:Image source="{resourceManager.getClass('RegistrationForm', 'flag')}"/>
    <s:Form>
```

```
        <s:FormItem label="{resourceManager.getString('RegistrationForm','personname')}">
            <s:TextInput/>
        </s:FormItem>
       <s:FormItem label="{resourceManager.getString('RegistrationForm','street_address')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','city')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','state')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','zip')}">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
    <s:Button id="b1"
        label="{resourceManager.getString('RegistrationForm','submit_button')}"
        click="registrationComplete()"/>
</s:Application>
```

To use a class, such as a programmatic skin, as a resource, you use the `ClassReference()` directive in your properties file. The following example embeds the MyCheckBoxIcon_en_US class in the properties file:

```
CHECKBOXSKIN=ClassReference("MyCheckBoxIcon_en_US")
```

You can bind the value of the `getClass()` method for programmatic skins, as the following example shows:

```
<mx:CheckBox selected="true"
    selectedUpIcon="{resourceManager.getClass('bundle1','CHECKBOXSKIN')}"
    selectedDownIcon="{resourceManager.getClass('bundle1','CHECKBOXSKIN')}"
    selectedOverIcon="{resourceManager.getClass('bundle1','CHECKBOXSKIN')}"/>
```

For more information about the ResourceManager, see “About the ResourceManager” on page 2105.

Changing locales at run time with the ResourceManager

A common use of localization is to provide multiple locales for a single application, and to let the user switch the locale at run time. You can compile all possible locales into the application and then choose from among them. This solution is not very flexible because you can only select from locales that you added to the application at compile time. This solution can also lead to larger applications because the resource properties files and all of their dependencies must be compiled into the application.

You can also compile resource properties files into resource module SWF files, and then dynamically load those SWF files at run time. These modules provide all the resources for the locales. The advantage to this approach is that the application SWF file is smaller because the resources are externalized, but it requires an additional network request for each resource module that the client loads. For information on using resource modules, see “Using resource modules” on page 2105.

You change the locale at run time by changing the value of the ResourceManager’s `localeChain` property. This property takes an Array as its value. The Array’s first element is the current locale (such as en_US or es_ES).

To be able to change the locale at run time without using resource modules, you compile all the available locales into the application at compile time by including them as part of the `locale` option. This compiler option takes a comma-separated list of locales. If you add a second locale, such as es_ES, change the `locale` option to the following:

```
-locale=en_US,es_ES
```

The Flex application uses the list of locales in the `localeChain` property to determine precedence when getting values from resource bundles. If a value does not exist in the first locale in the list, the Flex application looks for that value in the next locale in the list, and so on.

Before compiling additional locales into an application, you must generate the framework resources for that locale, if they are not already created. You do this with the copylocale command-line utility. For more information, see "Adding new locales" on page 2095.

The order of the locales on the command line can be important. The application defaults to the first locale in the list if you do not specifically set the value of the ResourceManager's `localeChain` property when the application initializes.

The following example lets you select a new locale from the ComboBox control. When you change that value, the application updates the `localeChain` property. The application's locale-specific assets, such as the form labels, flag image, and alert message should change to the new locale.

```
<?xml version="1.0"?>
<!-- resourcebundles/BasicLocaleChain.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        import mx.controls.Alert;
        [Bindable]
        private var locales:Array = [ "es_ES","en_US" ];
        private function initApp():void {
        b1.setStyle("downSkin", resourceManager.getClass("RegistrationForm", "flag"));

            // Initialize the ComboBox to the first locale in the locales Array.
            localeComboBox.selectedIndex = locales.indexOf(resourceManager.localeChain[0]);
        }
        private function registrationComplete():void {
            Alert.show(resourceManager.getString('RegistrationForm', 'thanks'));
        }
        private function comboChangeHandler():void {
            // Set the localeChain to either the one-element Array
            // [ "en_US" ] or the one-element Array [ "es_ES" ].
            resourceManager.localeChain = [ localeComboBox.selectedItem ];

            // This style is not bound to the resource bundle, so it must be reset when
            // the new locale is selected.
        b1.setStyle("downSkin", resourceManager.getClass("RegistrationForm", "flag"));
        }
    ]]></fx:Script>
    <fx:Metadata>
        [ResourceBundle("RegistrationForm")]
    </fx:Metadata>
    <s:Image source="{resourceManager.getClass('RegistrationForm', 'flag')}"/>
    <mx:ComboBox id="localeComboBox"
        dataProvider="{locales}"
        change="comboChangeHandler()"/>
    <s:Form>
```

```
        <s:FormItem label="{resourceManager.getString('RegistrationForm','personname')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','street_address')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','city')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','state')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','zip')}">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
    <s:Button id="b1"
        label="{resourceManager.getString('RegistrationForm','submit_button')}"
        click="registrationComplete()"/>
</s:Application>
```

When you compile this example application, you must add both locales to the `locale` option, as the following list of compiler options shows:

```
-locale=en_US,es_ES -allow-source-path-overlap=true -source-path=locale/{locale}
```

Because this application uses multiple locales, you must prepare properties files for each one. You should have the following files in your project to use this example:

```
/main/BasicLocaleChain.mxml
/main/locale/en_US/RegistrationForm.properties
/main/locale/en_US/images/unitedstates.gif
/main/locale/es_ES/RegistrationForm.properties
/main/locale/es_ES/images/spain.gif
```

The contents of the properties files for this example should be similar to the following:

```
# /locale/en_US/RegistrationForm.properties
registration_title=Registration
submit_button=Submit Form
personname=Name
street_address=Street Address
city=City
state=State
zip=ZIP Code
thanks=Thank you for registering!
flag=Embed("images/unitedstates.gif")

# /locale/es_ES/RegistrationForm.properties
registration_title=Registro
submit_button=Enviar el formulario
personname=Nombre
street_address=Dirección de calle
city=Ciudad
state=Estado
zip=Código postal
thanks=¡Gracias por inscribirse!
flag=Embed("images/spain.gif")
```

If you do not bind the properties in your application to your resources, then those values are not updated when the locale changes.

When you compile an application for multiple locales, the ResourceManager's `localeChain` property is initialized to the locales specified by the `locale` compiler option. For example, if the `locale` option is `-locale=en_US,es_ES`, the application defaults to the English resources because en_US is listed first. You can override this default initial value at run time by specifying the value of the `localeChain` as an application parameter in the `flashVars` variable in the HTML template.

If you write your own HTML template, you can pass variables as `flashVars` properties in the `<object>` and `<embed>` tags. The following example specifies that the es_ES locale is the first in the `localeChain` property's Array:

```
<object id='mySwf'
    classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
    codebase='http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab'
    height='100%'
    width='100%'>
    <param name='src' value='BasicLocaleChain.swf'/>
    <param name='flashVars' value='localeChain=es_ES,en_US'/>
    <embed name='mySwf'
        src='FlashVarTest.swf'
        pluginspage='http://www.adobe.com/go/getflashplayer'
        height='100%'
        width='100%'
        flashVars='localeChain=es_ES,en_US'
    />
>
```

If you are using SWFObject 2, the default template that Flex uses, define and pass the `flashVars` object to the `embedSWF()` JavaScript method, as the following example shows:

```
<script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            flashvars.localeChain="es_ES,en_US"
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "TestProject";
            attributes.name = "TestProject";
            attributes.align = "middle";
            swfobject.embedSWF(
                "TestProject.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
</script>
```

This initializes the value of the `localeChain` property to `["es_ES", "en_US"]`.

For more information about using `flashVars` properties, see "Passing request data with flashVars properties" on page 229.

About the ResourceManager

The ResourceManager manages all resource bundles, regardless of their source. They can be loaded as resource modules, compiled into the application, or created programmatically.

The ResourceManager class is a Singleton. All components that extend UIComponent, Formatter, or Validator have a `resourceManager` property, which lets you access this manager. If you create other classes that need to use the ResourceManager, you can call the `ResourceManager.getInstance()` method to get a reference to it.

Whenever you set the value of the `localeChain` property, the ResourceManager dispatches a `change` event. This event causes values that are bound to resource properties to be updated. You can manually dispatch this event by calling the ResourceManager's `update()` method. The `change` event causes the following to occur:

- Binding expressions involving resource-access methods of ResourceManager such as `getString()` are updated.
- Components that extend UIComponent, Formatter, or Validator execute their `resourcesChanged()` method. This is how components such as CurrencyFormatter can update their default value for resource-backed properties such as `currencySymbol`. If you are writing another kind of class that needs to respond to resource changes, you can listen for `change` events from the ResourceManager.

The `localeChain` property is an Array so that the ResourceManager can support incomplete locales. For example, suppose you localize an application for English as spoken in India (by using the "en_IN" locale). Most of the resources are the same as for U.S. English (the en_US locale), so there is no reason to duplicate them all in the en_IN locale's resources. The en_IN locale's properties files need only to have the resources that differ between the en_US and en_IN locales. If the ResourceManager has bundles for both en_US and en_IN and you set the value of the `localeChain` property to `["en_IN", "en_US"]`, the ResourceManager searches for a resource first in the en_IN bundle. If the resource is not found there, then the ResourceManager searches for the resource in the en_US bundle. If the ResourceManager does not find the resource you specify in any locale, its methods return null.

The most commonly used method of the ResourceManager for resource access is the `getString()` method. However, the ResourceManager supports other data types. You can get resources typed as Numbers, Booleans, Uints, and ints by using other methods, as the following example shows:

```
resourceManager.getNumber("myResources", "PRICE"); // Returns a Number like 19.99.
resourceManager.getInt("myResources", "AGE"); // Returns an int like 21.
resourceManager.getUint("myResources", "COLOR"); // Returns a Uint like 0xFF33AA.
resourceManager.getBoolean("myResources", "SENIOR") // Returns the Boolean like true.
```

You can also use methods such as `getClass()`, `getString()`, `getObject()`, and `getStringArray()`.

Using resource modules

Resource modules are SWF files, separate from your application SWF file, that contain resources bundles for a single locale. Your application can preload one or more resource modules as it starts up, before it displays its user interface. It can also load resource modules later, such as in response to the user selecting a new locale. The ResourceManager interacts with all resource bundles the same, whether the bundles it manages were originally compiled into the application or loaded from resource modules.

Resource modules can be a better approach to localization than compile-time resources because you externalize the resource modules that can be loaded at run time. This creates a smaller application SWF file, but then requires that you load a separate SWF file for each resource module that you use. The result can be an increased number of network requests and an aggregate application size that is larger than if you compiled the locale resources into the application. However, if you have many locales, then loading them separately should save resources in the long run.

You are limited as to when you can use resources in resource modules during the application initialization sequence. The earliest that you can access a resource bundle in a resource module is during the application's preinitialize event handler. You should not try to access a resource at class initialization time or during preloading.

Creating resource modules

To create a resource module, you must do the following:

- Determine the required resource bundles
- Create the resource module SWF file

The following sections describe these tasks.

Determining the required resource bundles to include in a resource module

Before you can compile a resource module, you must know which resource bundles to put into it. In other words, you must know which resource bundles your application — and all of its framework classes — actually require. This includes not just the custom resource bundles that you create, but also the framework resource bundles that are required by the application.

To determine which resource bundles an application needs, you use the `resource-bundle-list` compiler option. When you compile an application, this option outputs a list of the needed bundles by examining the `[ResourceBundle]` metadata on all of the classes in your application.

The `resource-bundle-list` option takes a filename as its argument. This filename is where the compiler writes the list of bundles. On the Macintosh OS X, you must specify an absolute path for this option. On all other operating systems, you can specify a relative or absolute path.

When using the `resource-bundle-list` option, you must also set the value of the `locale` option to an empty string.

The following command-line example generates a resource bundle list for the application MyApp:

```
mxmlc -locale= -resource-bundle-list=myresources.txt MyApp.mxml
```

In this example, the compiler writes a file called myresources.txt. This file contains a list similar to the following:

```
bundles = RegistrationForm collections containers controls core effects skins styles
```

In Flash Builder, you add the `locale` and `resource-bundle-list` options to the Additional Compiler Arguments field on the Flex Compiler pane in the project's properties. On Windows, the output file's location is relative to the Flash Builder directory, not the project's directory. You can specify an absolute path instead. On Macintosh, the option must be an absolute path.

If you use custom resource bundles, those will be included in this list. In this example, the name of the resource properties file is RegistrationForm. The others are bundles containing framework resources.

You use this list to instruct the compiler which resource bundles to include when you compile a resource module.

Compiling a resource module

To compile a resource module, you must use the mxmlc command-line compiler. You cannot compile resource modules by using Flash Builder. The output is a SWF file that you can then load at run time.

To compile a resource module on the command line, use the following guidelines:

- Do not specify an MXML file to compile.
- Specify the `include-resource-bundles` option.
- Specify the locale by using the `locale` option.
- Add the locales to the source path.

The `include-resource-bundles` option takes a comma-separated list of resource bundles to include in the resource module. You generated this list of required resource bundles in "Determining the required resource bundles to include in a resource module" on page 2106.

When compiling a resource module, you must also specify the locale by using the `locale` option.

The following example compiles a resource module for the en_US locale:

```
mxmlc -locale=en_US
    -source-path=locale/{locale}
    -include-resource-bundles=RegistrationForm,collections,containers,controls,core,
     effects,skins,styles
    -output en_US_ResourceModule.swf
```

You can compile the resources for only a single locale into each resource module. As a result, you cannot specify more than one locale for the `locale` option.

You should use a common naming convention for all of your locales' resource modules. For example, specify the locale in the SWF file's name, but keep the rest of the file name the same for all locales. This is because when you load the resource module, you want to be able to dynamically determine which SWF file to load. In the previous example, the resource module for the en_US locale is named en_US_ResourceModule.swf. If you then generated a resource module for the es_ES locale, it would be named es_ES_ResourceModule.swf.

Loading resource modules at run time

To load a resource module at run time, you use the ResourceManager's `loadResourceModule()` method. After the resource module has finished loading, you set the value of the ResourceManager's `localeChain` property to the newly-loaded locale.

The ResourceManager's `loadResourceModule()` method asynchronously loads a resource module. It works similarly to the `loadStyleDeclarations()` method of the StyleManager, which loads style modules.

The `loadResourceModule()` method takes several parameters. The first parameter is the URL for the resource modules's SWF file. This is the only required parameter. The second parameter is `update`. You set this to `true` or `false`, depending on whether you want the resource bundles to immediately update in the application. For more information, see "Updating resource modules" on page 2110. The final two parameters are `applicationDomain` and `securityDomain`. These parameters specify the domains into which the resource module is loaded. In most cases, you should accept the default values (`null`) for these parameters. The result is that the resource module is loaded into child domains of the current domains.

The `loadResourceModule()` method returns an instance of the IEventDispatcher class. You can use this object to dispatch ResourceEvent types based on the success of the resource module's loading. You have access to the `ResourceEvent.PROGRESS`, `ResourceEvent.COMPLETE`, and `ResourceEvent.ERROR` events of the loading process.

You should not call the `loadResourceModule()` method and then immediately try to use the resource module because the resource module's SWF file must be transferred across the network and then loaded by the application. As a result, you should add a listener for the ResourceManager's `ResourceEvent.COMPLETE` event. When the `ResourceEvent.COMPLETE` event is dispatched, you can set the `localeChain` property to use the newly-loaded resource module.

The following example loads a resource module when the user selects the locale from the ComboBox control. It builds the name of the SWF file (in this case, it is either en_US_ResourceModule.swf or es_ES_ResourceModule.swf), and passes that file name to the `loadResourceModule()` method.

```
<?xml version="1.0"?>
<!-- resourcebundles/ResourceModuleApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        import mx.controls.Alert;
        import mx.events.ResourceEvent;
        [Bindable]
        private var locales:Array = [ "es_ES","en_US" ];
        private function initApp():void {
            /* Set the index to -1 so that the prompt appears
               when the application first loads. */
            localeComboBox.selectedIndex = -1;
        }
        private function registrationComplete():void {
            Alert.show(resourceManager.getString('RegistrationForm', 'thanks'));
        }
        private function comboChangeHandler():void {
            var newLocale:String = String(localeComboBox.selectedItem);
            /* Ensure that you are not loading the same resource module more than once. */
            if (resourceManager.getLocales().indexOf(newLocale) != -1) {
                completeHandler(null);
            } else {
                // Build the file name of the resource module.
                var resourceModuleURL:String = newLocale + "_ResourceModule.swf";

                var eventDispatcher:IEventDispatcher =
                    resourceManager.loadResourceModule(resourceModuleURL);
                eventDispatcher.addEventListener(ResourceEvent.COMPLETE, completeHandler);
            }
        }

        private function completeHandler(event:ResourceEvent):void {
            resourceManager.localeChain = [ localeComboBox.selectedItem ];
            /* This style is not bound to the resource bundle, so it must be reset when
               the new locale is selected. */
        b1.setStyle("downSkin", resourceManager.getClass("RegistrationForm", "flag"));
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Metadata>
        [ResourceBundle("RegistrationForm")]
    </fx:Metadata>
    <s:Image source="{resourceManager.getClass('RegistrationForm', 'flag')}"/>
    <mx:ComboBox id="localeComboBox"
        prompt="Select One..."
        dataProvider="{locales}"
        change="comboChangeHandler()"/>
    <s:Form>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','personname')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','street_address')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','city')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','state')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','zip')}">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>

    <s:Button id="b1"
        label="{resourceManager.getString('RegistrationForm','submit_button')}"
        click="registrationComplete()"/>

</s:Application>
```

To compile an application that uses only resource modules, do not specify any locales to be compiled into the application. You do this by setting the value of the `locale` option to an empty String. For example, on the command line, you can compile an application named MyApp.mxml with the following command:

```
mxmlc -locale= MyApp.mxml
```

In Flash Builder, you add the following option to the Additional Compiler Arguments field on the Flex Compiler pane in the project's properties:

```
-locale=
```

If you compile an application that uses both compiled-in resources and resource modules, then you specify the locale(s) for the compiled-in resources with the `locale` option.

You should try to avoid loading the same resource module more than once in the application. To do this, you can use the ResourceManager's `getLocales()` method. This method returns an Array of locales. You can compare this list against the resource module's locale that you are about to load.

You can find out which resource bundles exist for a specified locale by using the `getBundleNamesForLocale()` method.

You can get a reference to a particular locale's resource bundle by calling the `getResourceBundle()` method. Once you have a reference to a ResourceBundle, you can use a for-in loop to iterate over its content object. For more information, see "Enumerating resources" on page 2117.

Loading remote resource modules

Loading a remote resource module typically requires a crossdomain.xml file that gives the loading application permission to load the SWF file. You can do without a crossdomain.xml file if your application is in the local-trusted sandbox, but this is usually restricted to SWF files that have been installed as applications on the local machine. For more information about crossdomain.xml files, see "Using cross-domain policy files" on page 125.

Also, to use a remote resource module, you must compile the loading application with network access (have the `use-network` compiler option set to `true`, the default). If you compile and run the application on a local file system, you might not be able to load a remotely accessible SWF file.

Updating resource modules

The second parameter of the `loadResourceModule()` method is `update`. Set the `update` parameter to `true` to force an immediate update of the resource bundles in the application. Set it to `false` to avoid an immediate update of the resource bundles in the application. The resources are updated the next time you call this method or the `unloadResourceModule()` method with the `update` property set to `true`.

Each time you call the `loadResourceModule()` method with the `update` parameter set to `true`, Adobe Flash Player and AIR reapply all resource bundles in the application, which can degrade performance. If you load multiple resource modules at the same time, you should set the `update` parameter to `false` for all but the last call to this method. As a result, Flash Player and AIR only apply the resource bundles once for all new resource module SWF files rather than once for each new resource module SWF file.

Preloading resource modules at run time

You can load a resource module when the application starts up by calling the `loadResourceModule()` method from your application initialization code, and then specifying the value of the `localeChain` property after the module loads. This is useful if you have a default locale that you want all users to start the application with. However, you can also specify the locale that the application should load on startup by passing `flashVars` properties in the HTML wrapper. This lets you specify a locale based on some run time value such as the `Accept-Language` HTTP header or the `Capabilities.language` property in ActionScript.

The following table describes the `flashVars` properties that you pass to set the preloaded resource modules:

flashVars property	Description
`localeChain`	A comma-separated list of locales that initializes the `localeChain` property of the ResourceManager class. If the `localeChain` property is not explicitly set, then it is initialized to the list of locales for which the application was compiled, as specified by the `locale` compiler option.
`resourceModuleURLs`	A comma-separated list of URLs from which resource modules will be sequentially preloaded. Resource modules are loaded by the same class as RSLs, but are loaded after the RSLs. The URLs can be relative or absolute.

As with URL parameters, you must separate these values with an ampersand (&). You must also ensure that the values are URL encoded.

If you write your own HTML template, you can pass variables as `flashVars` properties in the `<object>` and `<embed>` tags. The following example specifies that the es_ES locale's resource module is preloaded when the application launches:

```
<object id='mySwf'
    classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
    codebase='http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab'
    height='100%'
    width='100%'>
    <param name='src' value='ResourceModuleApp.swf'/>
    <param name='flashVars'
        value='resourceModuleURLs=es_ES_ResourceModule.swf&localeChain=es_ES'/>
    <embed name='mySwf'
        src='ResourceModuleApp.swf'
        pluginspage='http://www.adobe.com/go/getflashplayer'
        height='100%'
        width='100%'
        flashVars='resourceModuleURLs=es_ES_ResourceModule.swf&localeChain=es_ES'
    />
>
```

If you are using the SWFObject 2 template that Flex uses by default, define and pass the `flashVars` object to the `embedSWF()` JavaScript method, as the following example shows:

```
<script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
      flashvars.resourceModuleURLs="es_ES_resourceModule.swf"
     flashvars.localeChain="es_ES"
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "TestProject";
            attributes.name = "TestProject";
            attributes.align = "middle";
            swfobject.embedSWF(
                "TestProject.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
</script>
```

Using a combination of compile-time resources and resource modules

You might have a default locale that you want all users of your application to start with. You can compile this locale's resources into the application. You can then load external resource modules if the user changes the locale.

To do this, you compile the resource module as you normally would. When you compile the application, rather than specifying an empty String for the `locale` option, you set it to the locale that you want to compile into the application. This compiled-in locale's resource bundles become the default bundles when the application starts. You can then load a resource module SWF file as you normally would at run time.

Unloading resource modules at run time

You can unload a resource module when you are no longer using it. For example, if the user changes to a different locale, you can unload the resource module for one locale after loading the resource module for the new locale. This can reduce the memory footprint used by the application.

To unload a resource module, you can use the ResourceManager's `unloadResourceModule()` method. This removes the resource module's SWF file from memory. Its resources cannot be used until that module is reloaded.

Resource module SWF files are cached by the browser just like any other SWF file. As a result, if you unload a resource module, and then later want to reload it, Flash Player and AIR will load it from the browser's cache rather than make another network request. If the browser's cache was cleared, though, then Flash Player and AIR will load the resource module's SWF file with a network request again.

You can unload individual resource bundles rather than the entire resource module. You do this with the ResourceManager's `removeResourceBundle()` method. This method takes a locale and the resource bundle's name, which you can access with the `getBundleNamesForLocale()` method. Rather than iterate over the resource bundle names, you can use the `removeResourceBundlesForLocale()` method, which removes all resource bundles for a locale.

Creating resource bundles at run time

You can edit or create resource bundles programmatically. The process for creating a resource bundle is as follows:

1. Create a new ResourceBundle with the `new` operator.
2. Set the resource key/value pairs on the `content` object of the new ResourceBundle.
3. Add the bundle to the ResourceManager with the `addResourceBundle()` method.
4. Call the ResourceManager's `update()` method to apply the new changes.

The following example creates a new resource bundle for the fr_FR locale with two values:

```
var moreResources:ResourceBundle = new ResourceBundle("fr_FR", "moreResources");
moreResources.content["OPEN"] = "Ouvrez";
moreResources.content["CLOSE"] = "Fermez";
resourceManager.addResourceBundle(moreResources, false);
resourceManager.update();
```

After you add the resource bundle to the ResourceManager, you can use the ResourceManager's methods to find the new resources; for example:

```
resourceManager.localeChain = [ "fr_FR" ];
trace(resourceManager.getString("moreResources", "OPEN")); // outputs "Ouvrez"
```

If you use the `addResourceBundle()` method to add another bundle with the same locale and bundle name, the new one will overwrite the old one. However, creating new bundle values does not update the application. You must also call the ResourceManager's `update()` method to update existing resource bundles.

The following example replaces some of the existing resources in the en_US locale's RegistrationForm resource bundle with new values:

```
<?xml version="1.0"?>
<!-- resourcebundles/CreateReplacementBundle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        import mx.controls.Alert;
        [Bindable]
        private var locales:Array = [ "es_ES","en_US" ];
        private function initApp():void {
            /* Initialize the ComboBox to the first locale in the locales Array. */
            localeComboBox.selectedIndex = locales.indexOf(resourceManager.localeChain[0]);
        }
        private function registrationComplete():void {
            Alert.show(resourceManager.getString('RegistrationForm', 'thanks'));
        }
        private function comboChangeHandler():void {
            /* Set the localeChain to either the one-element Array
               [ "en_US" ] or the one-element Array [ "es_ES" ]. */
            resourceManager.localeChain = [ localeComboBox.selectedItem ];
        }

        private function createReplacementBundle():void {
            var newRB:ResourceBundle = new ResourceBundle("en_US", "RegistrationForm");

            newRB.content["registration_title"] = "Registration Form";
            newRB.content["submit_button"] = "Submit This Form";
            newRB.content["personname"] = "Enter Your Name Here:";
            newRB.content["street_address"] = "Enter Your Street Address Here:";
            newRB.content["city"] = "Enter Your City Here:";
            newRB.content["state"] = "Enter Your State Here:";
            newRB.content["zip"] = "Enter Your ZIP Code Here:";

            resourceManager.addResourceBundle(newRB);
            resourceManager.update();
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Metadata>
        [ResourceBundle("RegistrationForm")]
    </fx:Metadata>
    <s:Image source="{resourceManager.getClass('RegistrationForm', 'flag')}"/>
    <mx:ComboBox id="localeComboBox"
        dataProvider="{locales}"
        change="comboChangeHandler()"/>
    <s:Form>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','personname')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','street_address')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','city')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','state')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','zip')}">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>

    <s:Button id="b1"
        label="{resourceManager.getString('RegistrationForm','submit_button')}"
        click="registrationComplete()"/>

    <s:Button id="b2"
        label="Change Bundle"
        click="createReplacementBundle()"/>

</s:Application>
```

You can also programmatically create a new locale and new resource bundles for that locale. However, you should only create a new locale programmatically if you compiled with that locale's framework resources, as described in "Adding new locales" on page 2095.

If you programmatically create your own bundles for a new locale but do not create any framework bundles for that locale prior to compiling the application, you will get run-time exceptions when the framework components try to access framework bundles for the new locale. For example, suppose you programmatically create bundles for the fr_FR locale but do not compile with the fr_FR framework bundles. If you then set the `localeChain` property to `["fr_FR"]`, you will get run-time exceptions when the framework components try to access framework resources for the fr_FR locale. As a result, if you plan on creating bundles for the fr_FR locale at run time, you should compile the application with the fr_FR framework bundles. You can then set the `localeChain` property to `["fr_FR"]` without getting run-time errors.

When programmatically creating bundles, you cannot put the class of a run-time loaded image into the bundle because its class did not exist at compile time. Classes representing images are created only at compile time, for embedded images. As a result, you should explicitly set the source of an image rather than get the source of that image from a resource bundle.

The following example creates a new bundle for the fr_FR locale at run time. It explicitly assigns the source of the image when the fr_FR locale is selected rather than loading it from the resource bundle because the image's class was not available at compile time.

```
<?xml version="1.0"?>
<!-- resourcebundles/CreateNewLocaleAndBundle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        import mx.controls.Alert;
        import flash.display.*;
        [Bindable]
        private var locales:Array;
        private function initApp():void {
            locales =  [ "es_ES","en_US" ];

            /* Initialize the ComboBox to the first locale in the locales Array. */
            localeComboBox.selectedIndex = locales.indexOf(resourceManager.localeChain[0]);

            updateFlag();
        }
        private function registrationComplete():void {
            Alert.show(resourceManager.getString('RegistrationForm', 'thanks'));
        }
        private function comboChangeHandler():void {
            /* Set the localeChain to either the one-element Array
               [ "en_US" ] or the one-element Array [ "es_ES" ]. */
            resourceManager.localeChain = [ localeComboBox.selectedItem ];
            updateFlag();
        }

        private var newRB:ResourceBundle;

        private function updateFlag():void {
            if (resourceManager.localeChain[0] == "fr_FR") {
                /* Explicitly change the value of the flagImage source when the
                   locale is fr_FR because there was no class at compile time. */
                flagImage.source = "../assets/france.gif";
            } else {
                /* Get the class from the resource bundle; this assumes that the classes
                   for all other locales were embedded in the resource bundles at
                   compile time. */
              flagImage.source = resourceManager.getClass('RegistrationForm', 'flag');
            }
        }

        private function createNewBundle():void {
            locales.push("fr_FR");
            newRB = new ResourceBundle("fr_FR", "RegistrationForm");
            newRB.content["registration_title"] = "La Forme d'Enregistration";
            newRB.content["submit_button"] = "Soumettez La Forme";
            newRB.content["personname"] = "Nom";
            newRB.content["street_address"] = "Rue";
            newRB.content["city"] = "Ville";
            newRB.content["state"] = "Etat";
            newRB.content["zip"] = "Code postal";
          newRB.content["thanks"] = "Merci de l'enregistrement!";
```

```
            updateFlag();
            resourceManager.addResourceBundle(newRB);
            resourceManager.update();
        }

        private function resetApp():void {
            resourceManager.removeResourceBundlesForLocale("fr_FR");
            initApp();
            resourceManager.update();
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Metadata>
        [ResourceBundle("RegistrationForm")]
    </fx:Metadata>
    <s:Image id="flagImage"/>
    <mx:ComboBox id="localeComboBox"
        dataProvider="{locales}"
        change="comboChangeHandler()"/>
    <s:Form>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','personname')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','street_address')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','city')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','state')}">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="{resourceManager.getString('RegistrationForm','zip')}">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
    <s:Button id="b1"
        label="{resourceManager.getString('RegistrationForm','submit_button')}"
        click="registrationComplete()"/>
    <mx:HRule width="100%" strokeWidth="1"/>

    <s:HGroup>
        <s:Button id="b2" label="Add New Bundle" click="createNewBundle();"/>
        <s:Button id="b3" label="Reset" click="resetApp();"/>
    </s:HGroup>
</s:Application>
```

Adding, removing, or changing individual resources

You can add, remove, or change individual resources within a resource bundle in ActionScript. To do this, you first get a reference to the ResourceBundle with the `getResourceBundle()` method. You pass the locale and the name of the resource bundle to this method. You then edit the `content` property of the ResourceBundle, which is an object with key/value pairs for its resources.

The following example adds a new resource to the RegistrationForm resource bundle:

```
var rb:ResourceBundle =
    ResourceBundle(resourceManager.getResourceBundle("en_US", "RegistrationForm"));
rb.content["cancel_button"] = "Cancel";
```

You can change an existing value by specifying the key, as the following example shows:

```
rb.content["cancel_button"] = "Quit";
```

To delete a resource from a resource bundle, you use the `delete` keyword, followed by the key in the `content` object. The following example deletes the `cancel_button` resource that was added in the previous example:

```
delete rb.content["cancel_button"];
```

Enumerating resources

You can enumerate all resources in a resource bundle by using a `forin` loop in ActionScript. To do this, you get a reference to the resource bundle, and then loop over the `content` object.

Methods that you might use when enumerating resources include `getLocales()`, `getBundleNamesForLocale()`, and `getResourceBundle()`.

The `getLocales()` method returns an Array such as `["en_US", "ja_JP"]` which contains, in no particular order, all of the locales for bundles that exist in the ResourceManager. The `getBundleNamesForLocale()` method returns an Array such as `["controls", "containers"]` which contains all of the bundle names for the specified locale. The `getResourceBundle()` method takes a locale and a bundle name and returns a reference to the specified resource bundle.

The following example prints the keys and values for all resource bundles in all locales in the ResourceManager. This includes the framework bundles for the locales, in addition to any custom resources such as RegistrationForm.properties.

```
<?xml version="1.0"?>
<!-- resourcebundles/EnumerateAllBundles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="enumerateBundles()">
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        import mx.controls.Alert;
        private function registrationComplete():void {
            /* Use the ResourceManager to set localized values in ActionScript. */
            Alert.show(resourceManager.getString('RegistrationForm', 'thanks'));
        }

        private function enumerateBundles():void {
            for each (var locale:String in resourceManager.getLocales()) {
                ta1.text += "****************************************\n";
                ta1.text += "locale: " + locale + "\n";
                ta1.text += "****************************************\n";
                for each (var bundleName:String in
resourceManager.getBundleNamesForLocale(locale)) {
                    ta1.text += "  -------------------------------------\n";
                    ta1.text += "  bundleName: " + bundleName + "\n";
                    var bundle:ResourceBundle =
                      ResourceBundle(resourceManager.getResourceBundle(locale, bundleName));
                    for (var key:String in bundle.content) {
                        ta1.text += "    -" + key + ":" + bundle.content[key]  + "\n";
                    }
                }
            }
        }
    ]]></fx:Script>
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Metadata>
        [ResourceBundle("RegistrationForm")]
    </fx:Metadata>
    <s:Form>
        <s:FormItem label="@Resource(key='personname', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='street_address', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='city', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='state', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
        <s:FormItem label="@Resource(key='zip', bundle='RegistrationForm')">
            <s:TextInput/>
        </s:FormItem>
    </s:Form>
    <s:Button id="b1"
        label="@Resource(key='submit_button', bundle='RegistrationForm')"
        click="registrationComplete()"/>

    <s:TextArea id="ta1" width="100%" height="100%"/>
</s:Application>
```

Preventing memory leaks in modules and sub-applications

The ResourceManager is a singleton that holds a reference to all resource bundles in the system. This includes resource bundles used by sub-applications and modules. When a sub-application or module defines a new resource bundle that is not in the application domain of the main application, the ResourceManager creates a reference from the main application to the application domain of the sub-application or module. Because of this reference, the resource bundle cannot be garbage collected, which can result in extra memory being allocated but not used.

To avoid this situation, set the `addResourceBundle()` method's `useWeakReference` parameter to `true` in the sub-application or module. This creates a weak reference between the ResourceManager and the resource bundle, so that the resource bundle can be garbage collected. If the `useWeakReference` property is set to `false` (the default), the resource bundle will not be garbage collected. To prevent the resource bundle from being garbage collected until the right time, create a hard reference to it; remove that reference when you are ready remove the resource bundle or the module that uses it.

Custom formatting with resource bundles

The way applications present dates, times, and currencies varies greatly for each locale. For example, the U.S. standard for representing dates is month/day/year, whereas the European standard for representing dates is day/month/year. One of the more common localization tasks is to provide the formatting values for times, dates, and currencies.

The Spark formatters support the formats of all locales supported by the client's operating system. However, in some cases, you might want to override the behavior of these controls or provide your own fallback mechanism. You can use values in the resource bundles to set properties on Flex controls such as the DateTimeFormatter and CurrencyFormatter.

The following properties file sets the values that the DateTimeFormatter and CurrencyFormatter will use for the en_US locale:

```
# locale/en_US/FormattingValues.properties
CHROMECOLOR=0x0000FF
DATE_FORMAT=MM/dd/yy
TIME_FORMAT=L:NN A
CURRENCY_PRECISION=2
CURRENCY_SYMBOL=$
THOUSANDS_SEPARATOR=,
DECIMAL_SEPARATOR=.
```

For the British locale, the properties file might appear as follows:

```
# locale/es_ES/FormattingValues.properties
CHROMECOLOR=0xFF0000
DATE_FORMAT=dd/MM/yy
TIME_FORMAT=HH:NN
CURRENCY_PRECISION=2
CURRENCY_SYMBOL=£
THOUSANDS_SEPARATOR=.
DECIMAL_SEPARATOR=,
```

The following example uses resources bundles to set up the formatters:

```
<?xml version="1.0"?>
<!-- resourcebundles/FormattingExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)"
    chromeColor="{resourceManager.getUint('FormattingValues', 'CHROMECOLOR')}">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <s:DateTimeFormatter id="dateFormatter"
           dateTimePattern="{resourceManager.getString('FormattingValues', 'DATE_FORMAT')}"/>
        <s:DateTimeFormatter id="timeFormatter"
        dateTimePattern="{resourceManager.getString('FormattingValues', 'TIME_FORMAT')}"/>
        <s:CurrencyFormatter id="currencyFormatter"
            useCurrencySymbol="true"
            useGrouping="true"
          currencySymbol="{resourceManager.getString('FormattingValues', 'CURRENCY_SYMBOL')}"
            groupingSeparator="{resourceManager.getString('FormattingValues',
'THOUSANDS_SEPARATOR')}"
            decimalSeparator="{resourceManager.getString('FormattingValues',
'DECIMAL_SEPARATOR')}"/>
    </fx:Declarations>
    <fx:Script><![CDATA[
        import mx.resources.ResourceBundle;
        [Bindable]
        private var locales:Array = [ "en_US","es_ES"];
        [Bindable]
        private var dateValue:String;
        [Bindable]
        private var timeValue:String;
        [Bindable]
        private var currencyValue:String;
        private var d:Date = new Date();
        private function initApp(e:Event):void {
       localeComboBox.selectedIndex = locales.indexOf(resourceManager.localeChain[0]);
            applyFormats(e);

            // Updating the localeChain does not reapply formatters. As a result, you must
            // apply them whenever the ResourceManager's change event is triggered.
            resourceManager.addEventListener(Event.CHANGE, applyFormats);

        }
        private function comboChangeHandler():void {
            // Changing the localeChain property triggers a change event, so the
            // applyFormats() method will be called whenever you select a new locale.
            resourceManager.localeChain = [ localeComboBox.selectedItem ];
        }

        private function applyFormats(e:Event):void {
            dateValue = dateFormatter.format(d);
            timeValue = timeFormatter.format(d);
            currencyValue = currencyFormatter.format(1000);
        }
    ]]></fx:Script>
    <fx:Metadata>
        [ResourceBundle("FormattingValues")]
    </fx:Metadata>
    <mx:ComboBox id="localeComboBox"
        dataProvider="{locales}"
        change="comboChangeHandler()"/>

    <s:Form>
        <s:FormItem label="Date">
            <s:TextInput id="ti1" text="{dateValue}"/>
        </s:FormItem>
        <s:FormItem label="Time">
            <s:TextInput text="{timeValue}"/>
        </s:FormItem>
        <s:FormItem label="Currency">
            <s:TextInput text="{currencyValue}"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Adding styles and fonts to localized resources

To localize styles, you can bind the value of the style property to a value from the resource bundle. For example, if the resource property file defines the value of a `FONTCOLOR` key (`FONTCOLOR=0xFF0000`), you can use it by binding its value to the style property in your application, as the following examples show:

```
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    chromeColor="{resourceManager.getUint('MyStyles', 'color')}">
```

If you are using the ResourceManager to reference the value in the resource bundle, then be sure to use the appropriate method. For example, if the value is a color like 0xFF0000, use the ResourceManager's `getUint()` method. If the value is a class, like a programmatic skin, then use the `getClass()` method.

For color styles, you must use hex notation (such as 0xFF0000) rather than the VGA color names (such as red, blue, or fuchsia) in the resource properties files when binding a color style to a resource.

You can also localize the fonts that an application uses. For example, if the locale is ja_JP, then you would want to use a font that supports Japanese characters, but if the locale is en_US, then a font that supports the much smaller English character set would be adequate.

One approach to localizing fonts in your applications is to embed all the fonts and then get the name of the font's class selector from the resource bundle. In the resource properties file, you could set the values of the font, as the following example shows:

```
# /locale/en_US/FontProps.properties
TEXTSTYLE=myENFont
# /locale/ja_JP/FontProps.properties
TEXTSTYLE=myJAFont
```

In your application's style sheet, you embed both fonts, as the following example shows:

```
<fx:Style>
    @font-face {
        src: url("ENFont.ttf");
        fontFamily: EmbeddedENFont;
}
    @font-face {
        src: url("JAFont.ttf");
        fontFamily: EmbeddedJAFont;
    }
    .myENFont{
        fontFamily: EmbeddedENFont;
    }
    .myJAFont{
        fontFamily: EmbeddedJAFont;
    }
</fx:Style>
```

In your MXML tags, you can use the `styleName` property to apply the appropriate class selector, as the following example shows:

```
<s:RichText styleName="{resourceManager.getString('FontProps', 'TEXTSTYLE')}"/>
```

Another approach to localizing fonts is to use style modules, and load them at run time based on which locale is selected. To do this, for each font/locale combination, you compile a style module that embeds the font and sets any necessary selectors. The style module could be as simple as the following:

```
@font-face {
    src:url("../assets/MyENFont.ttf");
    fontFamily: myFontFamily;
    advancedAntiAliasing: true;
}
.global {
       fontFamily: myFontFamily;
}
```

When you compile the style module's SWF file, you can name it anything you want, but if you give it a name that contains the locale, you can then use that name programatically in your application. For example, name the style module MyStyleSheet_en_US.swf for the en_US locale and MyStyleSheet_ja_JP.swf for the ja_JP locale.

In your application, where you handle the resource bundle update logic, you can include logic that unloads the existing style module and loads a new one based on the current locale. You could also add the name of the style sheet to your resource bundle and extract it from that when the locale changes.

For more information on using style modules, see "Loading style sheets at run time" on page 1547.

Accessible applications

Accessibility overview

You create accessible content by using accessibility features included with Flex, by taking advantage of ActionScript designed to implement accessibility, and by following recommended design and development practices.

Visually impaired users, for example, might rely on assistive technology such as screen readers, which provide an audio version of screen content, or screen magnifiers, which display a small portion of the screen at a larger size, effectively reducing the visible screen area. Hearing-impaired users might read text and captions in the document in place of audio content. Other considerations arise for users with mobility or cognitive impairments.

The following list of recommended practices is not exhaustive, but suggests common issues to consider. Depending on your audience's needs, additional requirements may arise.

Visually impaired users For visually impaired users, keep in mind the following design recommendations:

- Design and implement a logical tab order for the tabs.
- Design the document so that constant changes in content do not unnecessarily cause screen readers to refresh. For example, you should group or hide looping elements.
- Provide captions for narrative audio. Be aware of audio in your document that might interfere with a user being able to listen to the screen reader.
- Use percentage sizing so that your applications scale properly at smaller screen sizes. This allows users of screen magnifiers to see more of your application at one time. Also take into account that many visually impaired users run applications with lower screen resolutions than other users.
- Ensure that foreground and background colors contrast sufficiently to make text readable for people with low vision.
- Ensure that controls don't depend on the use of a specific pointer device, such as a mouse or trackball.
- Ensure that components are accessible by keyboard. All Flex components defined as accessible include keyboard navigation. For a list of these components and the available keyboard commands for each, see "Accessible components and containers" on page 2130.

Color-blind users For color-blind users, ensure that color is not the only means of conveying information.

Users with mobility impairment For users with mobility impairment, keep in mind the following design recommendations:

- Ensure that controls don't depend on the use of a specific pointer device.
- Ensure that components are accessible by keyboard. All Flex components defined as accessible include keyboard navigation. For a list of these components and the available keyboard commands for each, see "Accessible components and containers" on page 2130.

Hearing-impaired users For hearing-impaired users, ensure that you add captions to audio content.

Users with cognitive impairment For users with cognitive impairments, such as dyslexia, keep in mind the following design recommendations:

- Ensure an uncluttered, easy-to-navigate design.
- Provide graphical imagery that helps convey the purpose and message of the application. These graphics should enhance, not replace, textual or audio content.
- Provide more than one method to accomplish common tasks.

About worldwide accessibility standards

Many countries, including the United States, Australia, Canada, Japan, and countries in the European Union, have adopted accessibility standards based on those developed by the World Wide Web Consortium (W3C). W3C publishes *Web Content Accessibility Guidelines*, a document that prioritizes actions that designers should take to make web content accessible. For information about the Web Accessibility Initiative, see the W3C website at www.w3.org/WAI.

In the United States, the law that governs accessibility is commonly known as Section 508, which is an amendment to the U.S. Rehabilitation Act. Section 508 prohibits federal agencies from buying, developing, maintaining, or using electronic technology that is not accessible to those with disabilities. In addition to mandating standards, Section 508 lets government employees and the public sue agencies in federal court for noncompliance.

For additional information about Section 508, see the U.S. government-sponsored website at www.section508.gov.

Viewing the Flex Accessibility web page

This topic contains an introduction to the accessibility features in Flex and to developing accessible applications. For the latest information on creating and viewing accessible Flex content, including supported platforms, known issues, screen reader compatibility, articles, and accessible examples, see the Flex Accessibility web page at www.adobe.com/go/flex_accessibility.

About screen reader technology

A screen reader is software designed to navigate through a website and read the web content aloud. Visually impaired users often rely on this technology. You can create content designed for use with screen readers for Microsoft® Windows® platforms only. Users who view your content must have Adobe® Flash® Player 9 or later, and Firefox or Internet Explorer on Windows 2000 or Windows XP or later.

JAWS, from Freedom Scientific, is one example of screen reader software. You can access the JAWS page on the Freedom Scientific website at www.hj.com/fs_products/software_jaws.asp. Another commonly used screen reader program is Window-Eyes, from GW Micro. To access the latest information on Window-Eyes, visit the GW Micro website at www.gwmicro.com.

***Note:** Flex support is most comprehensive in the JAWS screen reader. JAWS version 10 or later is strongly recommended.*

Screen readers help users understand what is contained in a web page or Flex application. Based on the keyboard shortcuts that you define, you can let users easily navigate through your application by using the screen reader.

Because different screen reader applications use various methods to translate information into speech, your content will vary in how it's presented to each user. As you design accessible applications, keep in mind that you have no control over how a screen reader behaves. You can only mark up the content in your applications so that you expose the text and ensure that screen reader users can activate the controls. This means that you can decide which objects in the application are exposed to screen readers, provide descriptions for the objects, and decide the order in which the objects are exposed to screen readers. However, you cannot force screen readers to read specific text at specific times or control the manner in which that content is read.

To use a JAWS screen reader with a Flex application, users must download a set of scripts before invoking the Flex application. For more information, see "Configuring a JAWS screen reader for Flex applications" on page 2129.

Flash Player and Microsoft Active Accessibility

Adobe Flash Player uses Microsoft Active Accessibility (MSAA), which provides a descriptive and standardized way for applications and screen readers to communicate. MSAA is available for Windows operating systems only. For more information on Microsoft accessibility technology, visit the Microsoft accessibility website at www.microsoft.com/enable/.

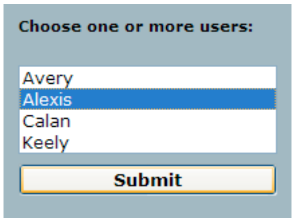
The Flash Player ActiveX control for Internet Explorer on Windows version 9 and later supports MSAA. The Flash Player plug-in for Firefox on Windows version 9.0.115 and later also supports MSAA. All versions of Windows standalone players do not support MSAA.

Flex supports a debugger version of Flash Player that can display debugging information during run time and generate profiling information so that you can more easily develop applications. However, the debugger version of Flash Player does not support accessibility.

Important: *MSAA is currently not supported in the opaque windowless and transparent windowless modes. (These modes are options in the HTML Publish Settings panel, available for use with the Windows version of Internet Explorer 4.0 or later, with the Flash ActiveX control.) If your content must be accessible to screen readers, avoid using these modes.*

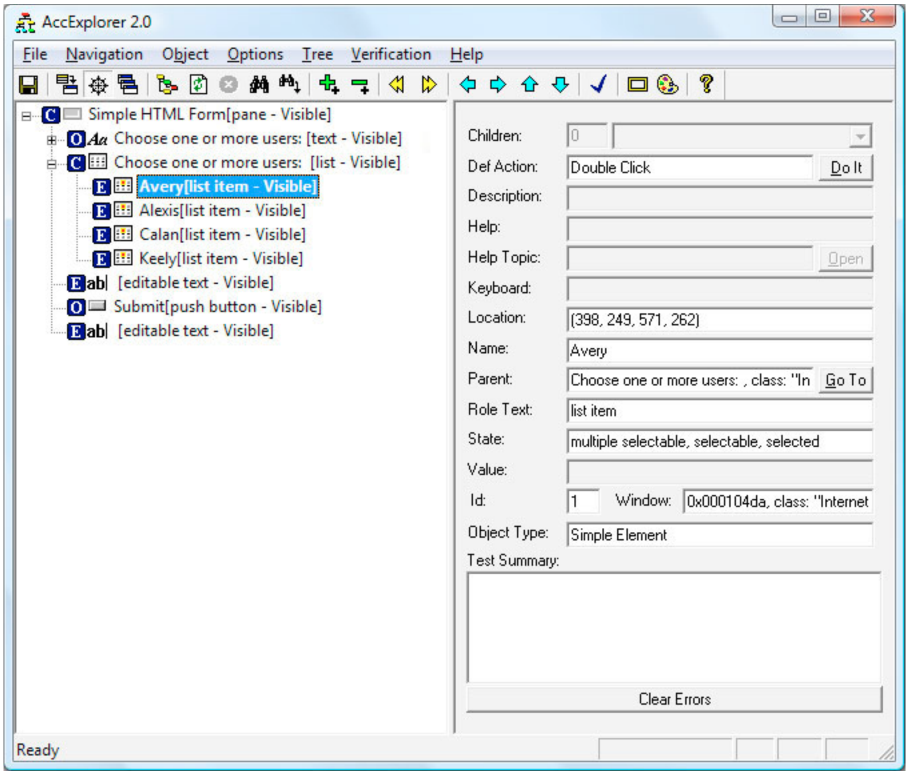
How MSAA works

MSAA is essentially a middleware architecture that conveys an object model of an application. This is most easily explained by an example. The following screenshot shows an HTML form, as displayed in a browser:



HTML form in a browser

Here is a screen capture that shows the object tree exposed by the browser to assistive technology:



Simple HTML Form[pane - Visible]
Choose one or more users: [text - Visible]
Choose one or more users: [list - Visible]
Avery[list item - Visible]
Alexis[list item - Visible]
Calan[list item - Visible]
Keely[list item - Visible]
[editable text - Visible]
Submit[push button - Visible]
[editable text - Visible]

Children: 0
Def Action: Double Click
Description:
Help:
Help Topic:
Keyboard:
Location: (398, 249, 571, 262)
Name: Avery
Parent: Choose one or more users: , class: "In
Role Text: list item
State: multiple selectable, selectable, selected
Value:
Id: 1 Window: 0x000104da, class: "Internet
Object Type: Simple Element
Test Summary:

Accessibility object tree

And finally, here is how JAWS for Windows 9.0, a popular screen reader for Windows, reads this form after loading the page in Internet Explorer:

```
Simple HTML Form.
Choose one or more children:
Extended select list box:
Avery, one of four.
Submit Button.
```

This example shows a simple, basic screen reader operation. A screen reader is a very complicated program that does much more than announce page contents; it speaks words under the mouse cursor, provides hundreds of keyboard shortcuts, and keeps track of user state in myriad ways.

MSAA under the surface

When Internet Explorer or Firefox loads a new page, it sends an event notification to the MSAA system. A screen reader, as an MSAA client, has registered with MSAA to receive event notifications, so the screen reader becomes aware that the browser has changed its object tree. The screen reader sends a special window message (WM_GETOBJECT) to the browser, asking for its root object.

The browser responds by sending back a pointer to a COM interface of type IAccessible. This interface has methods that allow the screen reader to ask for object properties, such as type, name, description, value, and visibility.

IAccessible also has methods for traversing the object tree in various ways—abstractly, in terms of tab order, in terms of geometric relationship, or in terms of pixel locations. Finally, IAccessible allows clients to manipulate UI objects programmatically, pressing buttons and altering selection and focus. The screen reader now traverses the tree of IAccessible objects, building up a tree of COM pointers.

For each object discovered, the screen reader determines the type of object (is it a container? button? textfield? and so on), the object's name, and possibly a few other properties. The screen reader now has enough information to construct its narrative text and feed it into the text-to-speech engine.

The browser, in order to be a proper MSAA server, has two main responsibilities. It must implement the IAccessible interface for every significant onscreen object that it displays, and it must notify the MSAA system whenever any part of the object tree changes. The Flash Player handles these two responsibilities for Flash content.

Accessible object model in Flash Player

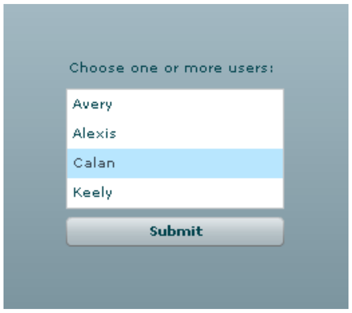
The Flash Player's accessible object tree consists of a single top-level container (the root accessible object) containing a flat collection of the following types of objects:

- **Text**: Regions of dynamic or static text in the movie. The principal property of a text object is its name, which, in keeping with MSAA convention, is equal to the contents of the text string being displayed. A text object may also have an associated description string. The Flash Player also makes an attempt to deduce labeling relationships between text and input text fields (text immediately above or to the left of an input text field is be taken as a label for that field), and between text and buttons (text entirely inside a button will be taken as a label for that button). Any text that is deduced to be a label will be omitted from the accessible object tree, and the content of that text will be used as the name of the object that it labels. Labels are never assigned to buttons or text fields that have author-supplied names.
- **Input text fields**: An input text object has a value, and optionally a name, a description string and a keyboard shortcut string. As described above, an input text object's name may come from a text object that is deduced to label it.
- **Buttons**: A button object has a state (pressed or not pressed), supports a programmatic default action that causes the button to depress momentarily, and may optionally have a name, a description string, and a keyboard shortcut string. As described above, a button object's name may come from a text object that is deduced to label it. Movie clips used as buttons in Flash Player are described by the player to MSAA as buttons, not as movie clips. The child objects inside buttons are note normally looked at by the player for accessibility purposes—buttons are "leaves" in the accessible object tree. There is one exception: for buttons that do not have author-supplied names, the player will look through the button's child elements (just one child-level deep) for a text element. If a text element is found, it is taken as a label for the button.
- **Simple movie clips**: All non-scripted movie clips at all levels are classified into one of two categories: simple or non-simple. A movie clip is simple if it does not have any other accessible objects (text, input text, buttons, or components) as children at any level of depth. Non-simple clips are not given any representation in the object tree; instead, their accessible-object contents are promoted into the flat top-level list of accessible objects. Simple clips, on the other hand, do show up in the object tree, expressed to MSAA as an image or animation widget. If a simple clip contains other simple clips, only the topmost simple clip is included in the object tree. A simple clip has a state (animated or not animated), and may optionally have a name and a description string. Note that all video regions are also treated as simple movie clips.
- **Scripted movie clips**: Flex (and Flash Professional) UI components present a special problem for the accessible object model in Flash Player. They are movie clips or sprites built from collections of text, input text, shapes, and buttons, but if Flash Player exposed those individual objects to MSAA, the meaning of the overall component would be lost. For this reason, a Flex UI (or Flash Professional) component may require a custom accessibility implementation. A movie clip or sprite that provides its own accessibility implementation is called a scripted movie clip, and it appears in the accessible object tree. The details of the accessibility implementation depend on the kind of component being implemented and are up to the component developer.

The Flash Player provides the flash.accessibility.AccessibilityImplementation class as a base class for creating custom accessibility implementations for Flex UI components in ActionScript 3. In Flex, this class is extended by mx.accessibility.AccImpl which serves as the base class for accessibility implementations in Flex components. A new accessibility implementation can be created by extending mx.accessibility.AccImpl for each new component.

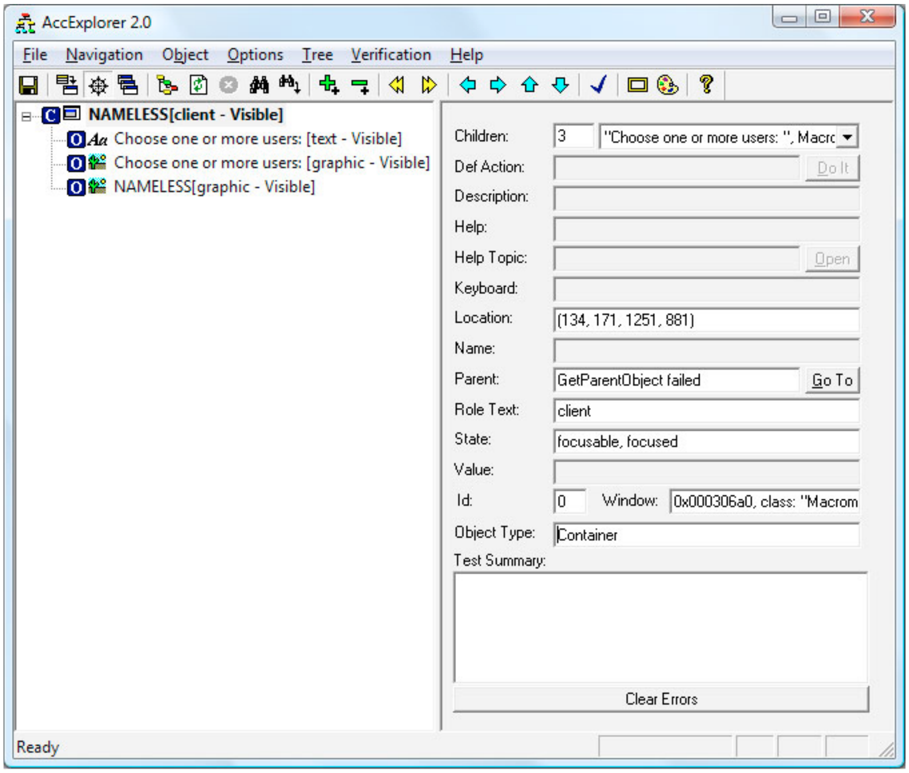
When a Flex project is compiled with Generate accessible SWF file set to true, the Flex compiler automatically enables accessibility implementations for the components in the project that have them. At runtime, when Flash Player discovers a component that has a custom accessibility implementation, it will treat that component as a scripted movie clip.

The following example shows a screenshot of a simple Flex form:



A simple Flex form

When the project is compiled with Generate accessible SWF file set to false, the object tree exposed by the browser to assistive technology contains only text and simple movie clips, as shown in the following image:

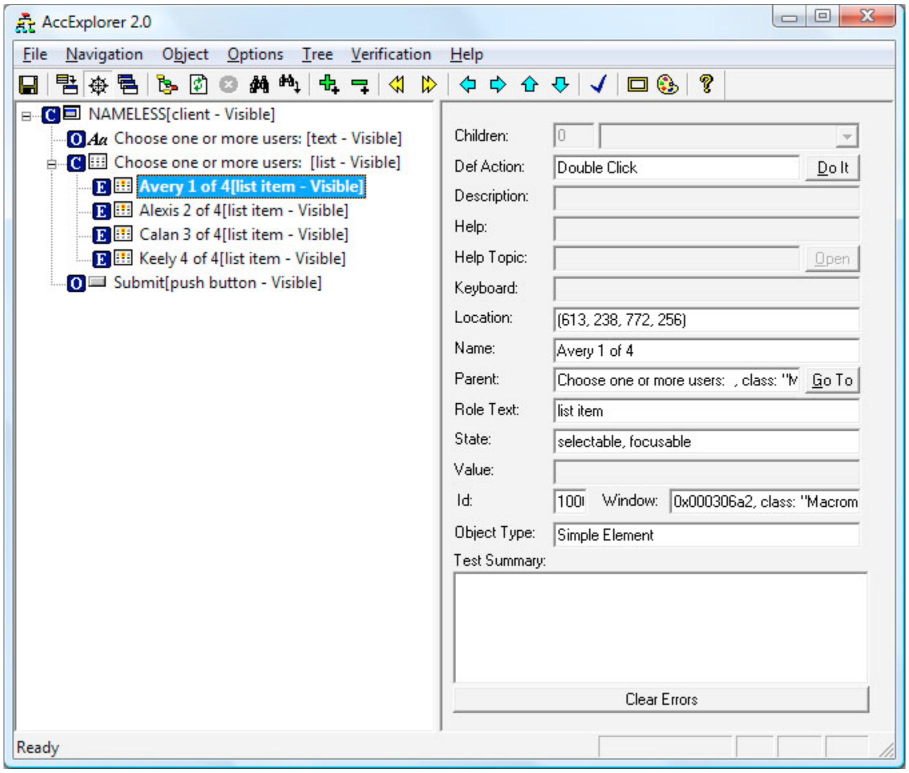


NAMELESS[client - Visible]
Choose one or more users: [text - Visible]
Choose one or more users: [graphic - Visible]
NAMELESS[graphic - Visible]

Children: 3 "Choose one or more users: ", Macrc
Def Action:
Description:
Help:
Help Topic:
Keyboard:
Location: (134, 171, 1251, 881)
Name:
Parent: GetParentObject failed
Role Text: client
State: focusable, focused
Value:
Id: 0 Window: 0x000306a0, class: "Macrom
Object Type: Container
Test Summary:

Accessibility object tree for a nonaccessible Flex application

When the same project is compiled with Generate accessible SWF file set to true, the object tree exposed by the browser to assistive technology includes scripted movie clips with accessibility implementations that properly expose them as text, a list box, and a button component:



NAMELESS[client - Visible]
Choose one or more users: [text - Visible]
Choose one or more users: [list - Visible]
Avery 1 of 4[list item - Visible]
Alexis 2 of 4[list item - Visible]
Calan 3 of 4[list item - Visible]
Keely 4 of 4[list item - Visible]
Submit[push button - Visible]

Children: 0
Def Action: Double Click
Description:
Help:
Help Topic:
Keyboard:
Location: (613, 238, 772, 256)
Name: Avery 1 of 4
Parent: Choose one or more users: , class: "M
Role Text: list item
State: selectable, focusable
Value:
Id: 100(Window: 0x000306a2, class: "Macrom
Object Type: Simple Element
Test Summary:

Accessibility object tree for an accessible Flex application

The methods of the flash.accessibility.AccessibilityImplementation class are a subset of the IAccessible interface, adapted slightly for ease of use. The Flash Player IAccessible interface for a scripted movie clip simply passes most calls from MSAA through to the flash.accessibility.AccessibilityImplementation subclass instance for that component. For example, when a screen reader needs to determine the default action of a mx.controls.Button component instance on the Stage in Flash Player, it calls the IAccessible method `get_accDefaultAction()` on the IAccessible interface exposed by the Flash Player for that component instance. The request is passed through to the `get_accDefaultAction()` method of the mx.accessibility.ButtonAccImpl accessibility implementation class instance, which returns the appropriate default action String for a button control, which is “Press.” The article Custom Accessibility Implementations explains in more detail how to build custom accessibility implementations for Flex components.

Configuring Flex applications for accessibility

Enabling accessibility in Flex

By default, Flex accessibility features are disabled. When you enable accessibility, you enable the application to communicate with a screen reader.

You can use one of the following methods to enable accessibility:

- Enable accessibility by default for all Flex applications so that all requests return accessible content.

  To enable accessibility for all Flex applications, edit the flex-config.xml file to set the `accessible` property to `true`, the default value, as the following example shows:

  ```
  <compiler>
      ...
      <accessible>true</accessible>
      ...
  <compiler>
  ```

- Enable accessibility in Adobe Flash Builder project properties.

  If you are using Adobe Flash Builder to develop Flex applications, you can enable accessibility by selecting the Generate accessible SWF file option in the Flex Compiler section of the project properties dialog. When you select this option, the compiler node within the .actionScriptProperties file located in the project directory is modified to generate accessible SWF files:

  ```
  <compiler additionalCompilerArguments="-locale en_US"
      generateAccessible="true">
  ```

- Enable accessibility by using the mxmlc command-line compiler.

  When you compile a file by using the mxmlc command-line compiler, you can use the `-accessible` option to enable accessibility, as the following example shows:

  ```
  mxmlc -accessible c:/dev/myapps/mywar.war/app1.mxml
  ```

  For more information on the command-line compiler, see “Flex compilers” on page 2164.

  When you compile a file by using the mxmlc command-line compiler, you can use a configuration variable to notify the compiler to generate the SWF file with accessibility enabled. The command-line syntax for the mxmlc.exe compiler allows the addition of `–compiler.accessible` (or just `–accessible`) to enable accessibility, as shown in the following examples:

  ```
  mxmlc –compiler.accessible c:/dev/myapps/ appl.mxml
  ```

  ```
  mxmlc -accessible c:/dev/myapps/appl.mxml
  ```

If you are building applications for Adobe LiveCycle Data Services ES and are deploying the application as MXML files, you can enable accessibility on an individual request by setting the `accessible` query parameter to `true`, as the following example shows:

```
http://www.mycompany.com/myflexapp/app1.mxml?accessible=true
```

If you edited the flex-config.xml file to enable accessibility by default, you can disable it for an individual request by setting the `accessible` query parameter to `false`, as the following example shows:

```
http://www.mycompany.com/myflexapp/app1.mxml?accessible=false
```

For more information on the command-line compiler, see “Flex compilers” on page 2164.

Configuring a JAWS screen reader for Flex applications

To use the JAWS screen reader with a Flex application, users must download scripts from the Adobe accessibility website before invoking a Flex application. Screen readers work best when in Forms mode, which lets users interact directly with the Flex application. These scripts let users switch between Virtual Cursor mode and Forms mode by using the Enter key from almost anywhere within a Flex application. If necessary, users can exit Forms mode by using the standard JAWS keystrokes.

Users can download these scripts, and the installation instructions, from the Adobe website at www.adobe.com/go/flex_accessibility.

To verify that the Flex scripts for JAWS are correctly installed, users can press the Insert+Q keys when JAWS is running. If the scripts are installed correctly, users hear "Updated for Adobe Flex 4" in the voice response to this keystroke.

It is important that you direct users with visual impairments to the script download page so that they have the necessary scripts to use JAWS effectively.

Accessible components and containers

To accelerate building accessible applications, Adobe built support for accessibility into Flex MX components and containers. These components and containers automate many of the most common accessibility practices related to labeling, keyboard access, and testing. They also help ensure a consistent user experience across rich Internet applications.

Accessible Spark components

Flex comes with the following set of accessible components and containers in the Spark component set.

Component	Screen reader behavior
ButtonBar control	Use the Tab key to move focus among the button of the ButtonBar control. Press the Spacebar to activate a button of the ButtonBar control. To cancel activating a button, press the Tab key to move the focus off the button control before releasing the Spacebar.
Button control	Press the Spacebar to activate the Button control. To cancel activating a button, press the Tab key to move the focus off the Button control before releasing the Spacebar.
CheckBox control	Press the Spacebar to activate the check box items. For more information on keyboard navigation, see “CheckBox control user interaction” on page 659.
ComboBox control	For more information on keyboard navigation, see “Spark ComboBox control user interaction” on page 541.
DropDownList control	For more information on keyboard navigation, see “Spark DropDownList control user interaction” on page 536.
List control	Screen reader navigation is the same as for keyboard navigation. For more information on keyboard navigation, see “Keyboard navigation” on page 953. For information on creating data tips (tool tips for individual list elements), see “Displaying DataTips” on page 946. For information on creating scroll tips (tool tips that provide information while the user scrolls through a list), see “Displaying ScrollTips” on page 947.
NumericStepper control	The name of a NumericStepper is, by default, an empty string. When wrapped in a FormItem element, the name is the FormItem's label. To override this behavior, set the Number ic Stepper control’s `accessibilityName` property. The name of each child button is: "More" and "Less". For more information on keyboard navigation, see “User interaction” on page 697.
Panel container	Screen reader announces the panel title only when Forms mode is inactive.
RadioButton control	With one radio button selected within a group, press the Enter key to enter that group. Use the arrow keys to move between items in that group. Press the Down and Right Arrow keys to move to the next item in a group; press the Up and Left Arrow keys to move to a previous item in the group. When using a screen reader, select a radio button by using the Spacebar key. For more information on keyboard navigation, see “RadioButton user interaction” on page 712.

Component	**Screen reader behavior**
RadioButtonGroup control	Screen reader navigation is the same as for the RadioButton control.
RichEdiableText control	Use the Home key to move to the beginning of a line. Use the End key to move to the end of a line. Use the Control-Home to move to the beginning of the text. Use the Control-End key to move to the end of the text.
Slider controls	The name of a HSlider or VSlider is, by default, an empty string. When wrapped in a FormItem element, the name is the FormItem's label. To override this behavior, set the Slider's `accessibilityName` property. The name of each child of the Slider control is: • "Page left" for HSlider; "Page up" for VSlider. • Position". • "Page right" for HSlider; "Page down" for VSlider. For more information on keyboard navigation, see “Keyboard navigation” on page 693.
Spinner control	The name of a Spinner control is, by default, an empty string. When wrapped in a FormItem element, the name is the FormItem's label. To override this behavior, set the Spinner `accessibilityName` property. The name of each child button is: "More" and "Less". For more information on keyboard navigation, see “User interaction” on page 697.
TabBar control	The name of a TabBar is, by default, an empty string. When wrapped in a FormItem element, the name is the FormItem's label. To override this behavior, set the TabBar's `accessibilityName` property.The name of each tab in the TabBar is its label. The TabBar and its individual tabs accept focus. A tab is not automatically pressed when focus is changed by arrow keys. To select a focused tab, use the spacebar.
TextArea control	Use the Home key to move to the beginning of a line. Use the End key to move to the end of a line. Use the Control-Home to move to the beginning of the text. Use the Control-End key to move to the end of the text.
TextInput control	Use the Home key to move to the beginning of a line. Use the End Up key to move to the end of a line.

Component	**Screen reader behavior**
TitleWindow container	The name of a TitleWindow is, by default, the title that it displays. To override this behavior, set the TitleWindow's `accessibilityName` property. A TitleWindow does not accept focus. A screen reader announces the TitleWindow control only when Forms mode is inactive.
ToggleButton control	Press the Spacebar to activate the control. To cancel activating a button, press the Tab key to move the focus off the control before releasing the Spacebar. The name of a ToggleButton is, by default, the label that it displays. When wrapped in a FormItem element, this label will be combined with the FormItem's label. To override this behavior, set the ToggleButton's `accessibilityName` property. To provide two separate names for the different states of an icon based ToggleButton, separate both names by a comma in the `accessibilityName` property. For example, `accessibilityProperty="Mute,Unmute"` When using state specific names like this, the button does not expose the "pressed" state when pressed.
VideoPlayer control	The name of a VideoPlayer is, by default, "VideoPlayer". When wrapped in a FormItem element, the name is the FormItem's label. To override this behavior, set the controls `accessibilityName` property. The name of each child control is: • Play/Pause control: "Play" or "Pause" • Scrub control: "Scrub Bar" • Play time indicator: the displayed text • Mute control: "Muted" or "Not muted" • Volume control: "Volume Bar" • Full Screen control: "Full Screen" To override the names of these child controls, reskin the VideoPlayer and set the `accessibilityName` of the controls. Use the Tab key to move focus among the child controls. Use the Spacebar to activate the Play/Pause control and the Full Screen control. Use the arrow keys to modify the Scrub Bar and Volume Bar controls.

Accessible MX Components

Flex comes with the following set of accessible components and containers in the MX component set.

Component	**Screen reader behavior**
Accordion container	Press the arrow keys to move the focus to a different panel, and then use the Spacebar or Enter key to select that panel. Use the Page Up and Page Down keys to move between individual panels of the container. When a screen reader encounters an Accordion container, it indicates each panel with the word *tab*. It indicates the current panel with the word *active*. For more information on keyboard navigation, see “Accordion container Keyboard navigation” on page 640.
AdvancedDataGrid control	The AdvancedDataGrid control supports the accessibility features in the DataGrid and Tree controls and provides additional features. For more information, see “Accessibility for the AdvancedDataGrid control” on page 2135. For more information on standard keyboard navigation, see “Keyboard navigation” on page 1450.
Alert control	In Forms mode, the text in the Alert control and the label of its default button are announced. When not in Forms mode, the text in the Alert control is announced twice when you press the Down Arrow.
Button control	Press the Spacebar to activate the Button control. To cancel activating a button, press the Tab key to move the focus off the Button control before releasing the Spacebar. When a screen reader encounters a Button control, activation varies, depending on the screen reader. In JAWS 6.10, the Spacebar activates Button controls when Forms mode is active. When Forms mode is inactive, the Spacebar or Enter key can be used to activate Button controls.
CheckBox control	Press the Spacebar to activate the check box items. For more information on keyboard navigation, see “CheckBox control user interaction” on page 659.
ColorPicker control	Announced as "colorpicker combo box." Open by using Control+Down Arrow, and close by using Control+Up Arrow. When open, you can use the four arrow keys to move among the colors. When open, the Enter key sets the color value to the currently selected color, as will Control+Up Arrow. When open, the Escape key closes the drop-down area and resets the color value to the previously selected color value.
ComboBox control	For more information on keyboard navigation, see “ComboBox control user interaction” on page 961.
DataGrid control	Press the arrow keys to highlight the contents, and then move between the individual characters within that field. When using a screen reader in forms mode, use the Tab key to move between editable TextInput fields in the DataGrid control. For more information on keyboard navigation, see “MX DataGrid control user interaction” on page 972.
DateChooser control	Press the Up, Down, Left, and Right Arrow keys to change the selected date. Use the Home key to reach the first enabled date in the month and the End key to reach the last enabled date in a month. Use the Page Up and Page Down keys to reach the previous and next months. For more information on keyboard navigation, see “User interaction” on page 675.
DateField control	Use the Control+Down Arrow keys to open the DateChooser control and select the appropriate date. When using a screen reader in Forms mode, use the same keystrokes as for keyboard navigation. When a screen reader encounters a DateChooser control in Forms mode, it announces the control as “DropDown Calendar *currentDate*, to open press Control Down, ComboBox,” where *currentDate* is the currently selected date. For more information on keyboard navigation, see “User interaction” on page 675.
Form container	For information on keyboard navigation, see “Defining a default button” on page 592.

Component	**Screen reader behavior**
Image control	An Image control with a tool tip defined is read by a screen reader only when Forms mode is inactive. The Image control is not focusable in Forms mode, or by the keyboard.
Label control	A Label control is read by a screen reader when it is associated with other controls, or when the Forms mode is inactive. The Label control is not focusable in Forms mode, or by the keyboard.
LinkButton control	LinkButton control activation when using a screen reader varies, depending on the screen reader. In JAWS 6.10, press the Spacebar to activate a LinkButton control when in Forms mode. When Forms mode is inactive, use the Spacebar or Enter key to activate the control. For more information on keyboard navigation, see "LinkButton control user interaction" on page 696.
List control	Screen reader navigation is the same as for keyboard navigation. For more information on keyboard navigation, see "List control user interaction" on page 950. For information on creating data tips (tool tips for individual list elements), see "Displaying DataTips" on page 946. For information on creating scroll tips (tool tips that provide information while the user scrolls through a list), see "Displaying ScrollTips" on page 947.
Menu control	Screen reader navigation is the same as for keyboard navigation. For more information on keyboard navigation, see "Menu control user interaction" on page 1001.
MenuBar control	Screen reader navigation is the same as for keyboard navigation. For more information on keyboard navigation, see "Menu control user interaction" on page 1001.
Panel container	Screen reader announces the panel title only when Forms mode is inactive.
RadioButton control	With one radio button selected within a group, press the Enter key to enter that group. Use the arrow keys to move between items in that group. Press the Down and Right Arrow keys to move to the next item in a group; press the Up and Left Arrow keys to move to a previous item in the group. When using a screen reader, select a radio button by using the Spacebar key. For more information on keyboard navigation, see "RadioButton user interaction" on page 712.
RadioButtonGroup control	Screen reader navigation is the same as for the RadioButton control.
Slider control	In forms mode the control is announced along with the orientation of the control (left-right or up-down) and the current value. The control uses the following keys for a left-right slider: • Left Arrow / Right Arrow—move slider to lower/higher value. • Page Up / Page Down—move slider to top/ bottom of range. • Down Arrow/Up Arrow—move slider to lower/higher value. • Home/End—move slider to top/ bottom of range. The control uses the following keys for an up/down slider: • Down Arrow/Up Arrow—move slider to lower/higher value. • Home/End—move slider to top/ bottom of range.
TabNavigator container	When a screen reader encounters a TabNavigator container pane, it indicates each pane with the word *tab*. It indicates the current pane with the word *active*. When a pane is selected, the user moves to that panel by pressing the Enter key. Press the arrow keys to move the focus to a different panel, and then use the Spacebar or Enter key to select that panel. Use the Page Up and Page Down keys to move between individual panels of the container. For more information on keyboard navigation, see "TabNavigator container Keyboard navigation" on page 638.

Component	**Screen reader behavior**
Text control	A Text control is not focusable and is read by screen readers only when Forms mode is inactive.
TextArea control	Use the Home key to move to the beginning of a line. Use the End key to move to the end of a line. Use the PageUp key to move to the beginning of the text. Use the PageDown key to move to the end of the text.
TextInput control	Use the Home or Page Down key to move to the beginning of a line. Use the End or Page Up key to move to the end of a line.
TitleWindow container	A screen reader announces the TitleWindow control only when Forms mode is inactive.
ToolTipManager	When a screen reader is used, the contents of a tool tip are read after the item to which the tool tip is attached gets focus. Tool tips attached to nonaccessible components (other than the Image control) are not read.
Tree control	Press the Up and Down Arrow keys to move between items in a Tree control. To open a group, press the Right Arrow key or Spacebar. To close a group, press the Left Arrow key or Spacebar. For more information on keyboard navigation, see “Editing a node label at run time” on page 983and “Tree user interaction” on page 983.

Accessibility for the AdvancedDataGrid control

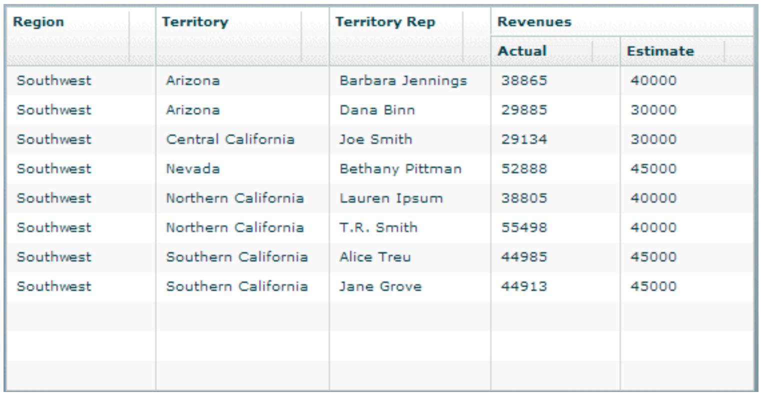
The AdvancedDataGrid control supports the accessibility features in the DataGrid and Tree controls and adds additional features to support accessibility.

Cell and header navigation

For AdvancedDataGrid header cells, each header cell is selectable and reported individually. Pressing the Up Arrow key when in the first row of the control shifts the focus to the header cell.

When focus is moved to a header cell, the screen reader reports: "From: Column 1 press space to sort ascending on this field. Press Control+space to add this field to sort."

The following figure shows an AdvancedDataGrid control that uses a column group for the Revenues column. If the control defines a column group, then the screen reader reports the header information as: "Revenues: Column 1, spans 2 columns."



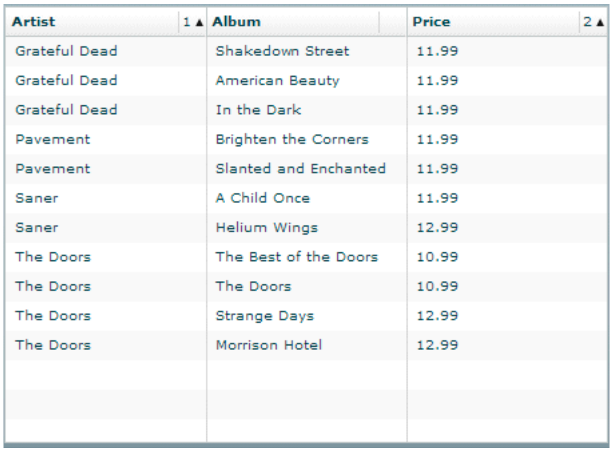
Region	Territory	Territory Rep	Revenues	
			Actual	Estimate
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
Southwest	Central California	Joe Smith	29134	30000
Southwest	Nevada	Bethany Pittman	52888	45000
Southwest	Northern California	Lauren Ipsum	38805	40000
Southwest	Northern California	T.R. Smith	55498	40000
Southwest	Southern California	Alice Treu	44985	45000
Southwest	Southern California	Jane Grove	44913	45000

The data reported for body cells depends on the setting of the `selectionMode` property of the AdvancedDataGrid control, as described in the following table:

Selection mode	**Screen reader output**
`singleRow`	Information corresponding to the entire row.
`singleCell`	Information corresponding to the selected cell only.

Example navigation of flat data

The following image shows an AdvancedDataGrid control displaying flat data:



Artist 1 ▲	Album	Price 2 ▲
Grateful Dead	Shakedown Street	11.99
Grateful Dead	American Beauty	11.99
Grateful Dead	In the Dark	11.99
Pavement	Brighten the Corners	11.99
Pavement	Slanted and Enchanted	11.99
Saner	A Child Once	11.99
Saner	Helium Wings	12.99
The Doors	The Best of the Doors	10.99
The Doors	The Doors	10.99
The Doors	Strange Days	12.99
The Doors	Morrison Hotel	12.99

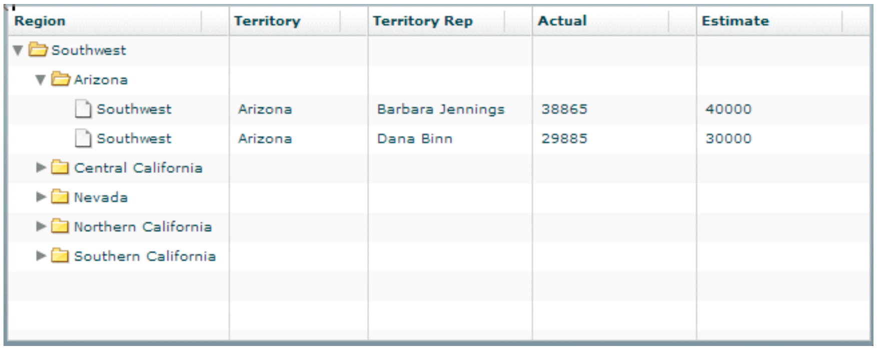
In this example:

- When you press the Tab key to move focus to the control, the screen reader reports: "ListBox."
- Pressing the Down Arrow and Up Arrow keys navigates the rows of the control.
    - When the `selectionMode` property is set to `singleRow`, the screen reader reports all the data in the row, for example: "Artist: Grateful Dead, Album: Shakedown Street, Price: 11.99."
    - When the `selectionMode` property is set to `singleCell`, the screen reader reports only the selected cell's data. For the cell in the first column of the row, the screen reader reports: "Artist: Grateful Dead, Row 1." The row information is reported only when focus is on the first column, not for other columns in the row. Therefore, for the second cell, the screen reader reports: "Album: ShakeDown Street."
- Pressing the Up Arrow key when the focus is on the first row of the control moves the focus to the header cell. The screen reader reports: "From: Column 1 press space to sort ascending on this field. Press control space to add this field to sort."
    - Pressing the Spacebar sorts the column.
    - Multicolumn sorting is supported. The sort order is reported in case of multicolumn sorting.

  For example, if the Album column is sorted and the sort order is 2, the screen reader reports: "Album: Column 2 sorted ascending, sort order 2 Press space to sort descending on this field. Press Control+space to add this field to sort."

Example navigation of hierarchical data

The following image shows an AdvancedDataGrid control displaying hierarchical data:



Region	Territory	Territory Rep	Actual	Estimate
▼ Southwest				
▼ Arizona				
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
▶ Central California				
▶ Nevada				
▶ Northern California				
▶ Southern California				

In this example:

- When you press the Tab key to move focus to the control, the screen reader reports: "Treeview."
- Pressing the Shift+Control+Right Arrow keys opens a node.
- Pressing the Down Arrow key moves the focus to the next row.

  If you move focus to the first row under Arizona, the screen reader reports: "Level2. Region: Southwest, Territory: Arizona, TerritoryRep: Barbara Jennings, Actual: 38865, Estimate: 40000, 1 of 2. Press Control+Shift+Right Arrow to open, Control+Shift+Left Arrow to close. Open."

Creating tab order and reading order

There are two aspects of tab indexing order—the *tab order* in which a user navigates through the web content, and the *reading order* in which things are read by the screen reader.

Flash Player uses a tab index order from left to right and top to bottom. However, if this is not the order you want to use, you can customize both the tab order and the reading order by using the InteractiveObject.tabIndex property. (In ActionScript, the `tabIndex` property is synonymous with the reading order.)

Tab order You can use the `tabIndex` property of every component to create a tab order that determines the order in which objects receive input focus when a user presses the Tab key.

Reading order You can use the `tabIndex` property to control the order in which the screen reader reads information about the object. To create a reading order, you assign a value to the `tabIndex` property for every component in your application. You should set the `tabIndex` property for every accessible object, not just the focusable objects. For example, you must set the `tabIndex` property for a Text control even though a user cannot tab to it. If you do not set the `tabIndex` property for every accessible object, Flash Player puts that object at the end of the tab order, rather than in its appropriate tab order location.

Scrolling to a component when tabbing

As a general rule, you should structure your application so that all components fit in the available screen area; otherwise you will have to adds vertical or horizontal scroll bars as necessary. Scroll bars let users move around the application to access components outside of the screen area.

If your application uses scroll bars, tabbing through the application components does not automatically scroll the application to make the currently selected component visible. You can add logic to your application to automatically scroll to the currently selected component, as the following example shows:

```
<?xml version="1.0"?>
<!-- accessibility\ScrollComp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="setupFocusViewportWatcher();">

    <fx:Script>
        <![CDATA[

        import mx.core.Container;
        import mx.core.EdgeMetrics;

        [Bindable]
        public var cards: Array = [
            {label:"Visa", data:1},
            {label:"Master Card", data:2},
            {label:"American Express", data:3} ];

        [Bindable]
        public var selectedItem:Object;

        [Bindable]
        public var forListDP:Array = [
            {label:'Apple', data:10.00},
            {label:'Banana', data:15.00},
            {label:'Melon', data:3.50},
            {label:'Kiwi', data:7.65},
            {label:'123', data:12.35 },
            {label:'some', data:10.01 }];

        // Set up the event listener for the focusIn event.
        public function setupFocusViewportWatcher():void {
            addEventListener("focusIn", makeFocusedItemVisible);
        }

        public function makeFocusedItemVisible(event:FocusEvent):void {
            // Target is the actual object that has focus.
            var target:InteractiveObject = InteractiveObject(event.target);

            // OriginalTarget is the component that has focus as some
            // component actually delegate true focus to an internal object.
            var originalTarget:InteractiveObject =
        InteractiveObject(focusManager.findFocusManagerComponent(target));

            // The viewable portion of a container
            var viewport:Rectangle = new Rectangle();
            do {
                // Cycle through all parents looking for containers.
                if (target.parent is Container) {
                    var viewportChanged:Boolean = false;
                    var c:Container = target.parent as Container;
                    // Get the viewable area in the container.
                    var vm:EdgeMetrics = c.viewMetrics;
                    viewport.x = vm.left;
                    viewport.y = vm.top;
                    viewport.width =
                        c.width / c.scaleX - vm.left - vm.right;
                    viewport.height =
                        c.height / c.scaleY - vm.top - vm.bottom;

                    // Calculate the position of the target in the container.
                    var topLeft:Point = new Point(0, 0);
                    var bottomRight:Point =
                new Point(originalTarget.width, originalTarget.height);
                    topLeft = originalTarget.localToGlobal(topLeft);
                    topLeft = c.globalToLocal(topLeft);
                    bottomRight = originalTarget.localToGlobal(bottomRight);
                    bottomRight = c.globalToLocal(bottomRight);

                    // Figure out if we have to move the scroll bars.
                    // If the scroll bar moves, the position of the component
                    // moves as well. This algorithm makes sure the top
                    // left of the component is visible if the component is
                    // bigger than the viewport.
                    var delta:Number;
                    if (bottomRight.x > viewport.right) {
                        delta = bottomRight.x - viewport.right;
                        c.horizontalScrollPosition += delta;
                        topLeft.x -= delta;
                        viewportChanged = true;
                    }
                    if (topLeft.x < viewport.left) {
                        // leave it a few pixels in from the left
                        c.horizontalScrollPosition -=
                            viewport.left - topLeft.x + 2;
                        viewportChanged = true;
                    }

                    if (bottomRight.y > viewport.bottom) {
                        delta = bottomRight.y - viewport.bottom;
                        c.verticalScrollPosition += delta;
                        topLeft.y -= delta;
                        viewportChanged = true;
                    }

                    if (topLeft.y < viewport.top) {
                        // leave it a few pixels down from the top
                        c.verticalScrollPosition -=
                            viewport.top - topLeft.y + 2;
                        viewportChanged = true;
                    }

                    // You must the validateNow() method to get the
                    // container to move the component before working
                    // on the next parent.
                    // Otherwise, your calculations will be incorrect.
                    if (viewportChanged) {
                        c.validateNow();
                    }
                }

                target = target.parent;
            }
        }
```

```
            while (target != this);
        }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <fx:Model id="statesModel" source="assets/states.xml"/>
    </fx:Declarations>
    <mx:Panel
            x="58" y="48"
            width="442" height="201"
            layout="absolute"
            title="Tab through controls to see if focus stays in view">

        <mx:VBox x="10" y="10" verticalScrollPolicy="off">
            <mx:TextInput/>
            <mx:TextInput/>
            <mx:TextArea width="328" height="64"/>
            <mx:ComboBox dataProvider="{cards}" width="150"/>
            <mx:DataGrid dataProvider="{forListDP}" />
            <mx:DateChooser yearNavigationEnabled="true"/>
            <mx:List id="source"
                width="75"
                dataProvider="{statesModel.state}"/>
        </mx:VBox>
    </mx:Panel>
</s:Application>
```

Creating accessibility with ActionScript

For accessibility properties that apply to the entire document, use the flash.accessibility.AccessibilityProperties class. For more information on this class, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

The following table lists some of the relevant properties of the AccessibilityProperties class.

Property	**Type**	**Description**
`description`	String	Specifies a description for the component that is read by the screen reader.
`forceSimple`	Boolean	Hides the children of a component from a screen reader when set to `true`. The default value is `false`.
`name`	String	Specifies a description of the component that is read by the screen reader. When accessible objects do not have a specified name, a screen reader uses a generic word, such as *Button*.
`shortcut`	String	Indicates a keyboard shortcut associated with this display object.
`silent`	Boolean	Hides a component from a screen reader when set to `true`. The default value is `false`.

Modifying these properties has no effect by itself. You must also use the `Accessibility.updateProperties()` method to inform screen reader users of Flash Player content changes. Calling this method causes Flash Player to reexamine all accessibility properties, update property descriptions for the screen reader, and, if necessary, send events to the screen reader that indicate changes occurred.

When updating the accessibility properties of multiple objects at once, you must include only a single call to the `Accessiblity.updateProperties()` method. (Excessive updates to the screen reader can cause some screen readers to become too verbose and can impact your application’s performance.)

Implementing screen reader detection with the Accessibility.isActive() method

To create content that behaves in a specific way if a screen reader is active, you can use the ActionScript Accessibility.active property, which is set to a value of `true` if a screen reader is present, and `false` otherwise. You can then design your content to perform in a way that is compatible with screen reader use, such as by hiding child elements from the screen reader.

For example, you could use the `Accessibility.active` property to decide whether to include unsolicited animation. *Unsolicited animation* means animation that happens without the screen reader doing anything. This can be very confusing for screen readers.

For more information on the Accessibility class, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Creating a custom accessibility implementation

Flex component developers need to be able to provide their own accessibility implementations for custom components. Flash Player provides the flash.accessibility.AccessibilityImplementation class as a base class for creating custom accessibility implementations for Flex and Flash UI components in ActionScript 3. In Flex, this class is extended by mx.accessibility.AccImpl which serves as the base class for accessibility implementations in Flex components.

You create a new accessibility implementation by extending mx.accessibility.AccImpl for each new component. The methods of the flash.accessibility.AccessibilityImplementation class are a subset of the IAccessible interface, adapted slightly to make life easier. The Flash Player's IAccessible interface for a scripted movie clip passes most calls from MSAA through to the flash.accessibility.AccessibilityImplementation subclass instance for that component. Accessibility implementations must also send event notifications to MSAA in a variety of situations. Event notifications are sent by using the `flash.accessibility.Accessibility.sendEvent()` method.

The way in which an accessibility implementation implements the IAccessible interface and the events that it sends depend on the kind of component being implemented. Consider two primary forms of guidance: First, the MSAA documentation has a list of guidelines for accessible object implementations according to component type. Second, an accessibility implementations's behavior should match as closely as possible the behavior exhibited by the equivalent form element (if any such element exists) in an HTML page inside Internet Explorer or Firefox or in the Windows operating system. This behavior can be examined using the AccExplorer, Inspect32, and AccEvent tools, available in the Microsoft Active Accessibility 2.0 SDK.

Collectively, these two forms of guidance constitute an MSAA model for IAccessible implementations. This model is loosely specified. The only definitive test of an IAccessible implementation is to test it with the latest versions of screen readers.

This article explains how to define a custom accessibility implementation for a component. For this example, we'll add an accessibility implementation to the mx.controls.PopUpButton and, by extension, the mx.controls.PopUpMenuButton class. First, create a new Flex project named “PopUpMenuButton.” Be sure to enable accessibility in the project using one of the methods explained in “Configuring Flex applications for accessibility” on page 2128. Follow along with the article to create a new implementation, or download the finished source files for the project.

For an overview of the PopUpButton and PopUpMenuButton controls, see “PopUpButton control” on page 702 and “PopUpMenuButton control” on page 1003. See also mx.controls.PopUpButton and mx.controls.PopUpMenuButton in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Add accessibility mix-ins to component class definition

Defining a custom accessiblity implementation for a component requires changes the component's class file. Copy the PopUpButton class file from its location in the Flex SDK source code, /sdks/4.6.0/frameworks/projects/framework/src/mx/controls/PopUpButton.as to a directory in the source folder for the project, /mx/controls/PopUpButton.as.

Next, copy the included files /sdks/4.6.0/frameworks/projects/framework/src/mx/core/Version.as to /mx/core/Version.as and /sdks/4.6.0/frameworks/projects/framework/src/mx/styles/metadata/IconColorStyles.as to /mx/styles/metadata/IconColorStyles.as so that the project compiles without errors. When the project is compiled, the Flex compiler will override the existing PopUpButton class contained within the Flex framework, with the modified class in project folder, which will support accessibility. Open the PopUpButton class file that you just copied over from the Flex SDK. At line 168 in the code, add the following meta tag:

```
[AccessibilityClass(implementation="mx.accessibility.PopUpButtonAccImpl")]
```

The `[AccessibilityClass]` meta tag lets the compiler know which class serves as the accessibility implementation for this component. Add the following code to create a placeholder for the static `createAccessibilityImplementation()` method at line 244, just before the PopUpButton constructor:

```
…
    //-------------------------------------------------------------------- ----
    //
    //  Class mixins
    //
    //--------------------------------------------------------------------           ----
    /**
     *  Placeholder for mixin by PopUpButtonAccImpl.
     */
     mx_internal static var createAccessibilityImplementation:Function;
…
```

This `createAccessibilityImplementation()` method will be assigned by the PopUpButtonAccImpl accessibility implementation class we'll create. At line 469, among the overridden methods from UIComponent class, add the following `initializeAccessibility()` method:

```
/**
 *  @inheritDoc
 */
  override protected function initializeAccessibility():void
    {
        if (PopUpButton.createAccessibilityImplementation != null)
        PopUpButton.createAccessibilityImplementation(this);
    }
```

The `initializeAccessibility()` method is invoked by the `UIComponent.initialize()` method to initialize accessibility for a component instance at runtime.

Create the accessibility implementation class definition

To define the PopUpButtonAccImpl class, copy the ButtonAccImpl class file from its location in the Flex SDK source code, /sdks/4.6.0/frameworks/projects/framework/src/mx/accessibility/Button.as, to a directory in the source folder for the project. Save the file with a new file name, /mx/accessibility/PopUpButtonAccImpl.as.Open the PopUpButtonAccImpl class file that you copied over from ButtonAccImpl class in the Flex SDK. Update the `import` statement block to import the PopUpButton, Menu, AccessibilityProperties and Rectangle classes:

```
...
package mx.accessibility
{
import flash.accessibility.Accessibility;
import flash.accessibility.AccessibilityProperties;
import flash.events.Event;
import flash.events.KeyboardEvent;
import flash.geom.Rectangle;
import flash.ui.Keyboard;
import mx.controls.Menu;
import mx.controls.PopUpButton;
import mx.core.UIComponent;
import mx.core.mx_internal;
...
```

At line 61 in the code, update the class definition to read:

```
...
    public class PopUpButtonAccImpl extends AccImpl
...
```

At line 93 in the code update the `hookAccessibility` method to read:

```
...
    private static function hookAccessibility():Boolean
    {
        PopUpButton.createAccessibilityImplementation =
            createAccessibilityImplementation;
        return true;
    }
...
```

The `hookAccessibility()` method defines the PopUpButton class's static `createAccessibilityImplementation()` method to use the method of the same name defined in the PopUpButtonAccImpl class. It is called once at runtime when the PopUpButtonAccImpl class initializes and defines the static `accessibilityHooked` property.Replace the `createAccessibilityImplementation()` method at line 149 with the following:

```
...
    mx_internal static function createAccessibilityImplementation(
                                                    component:UIComponent):void
    {
        component.accessibilityImplementation = new PopUpButtonAccImpl(component);
    }
...
```

The `createAccessibilityImplementation()` method assigns a new PopUpButtonAccImpl instance to the `accessibilityImplementation` property of a given component instance.Add the following four static state and role constants at line 122:

```
...
    /**
     * The object is hot-tracked by the mouse, which means that its appearance
     * has changed to indicate that the pointer is located over it.
     */
     private static const STATE_SYSTEM_HOTTRACKED:uint = 0x00000080;

    /**
     *  Object displays a pop-up menu or window when invoked.
     */
     private static const STATE_SYSTEM_HASPOPUP:uint = 0x40000000;

    /**
     * The role represents a button that has a drop-down list icon directly adjacent
     * to the button.
     */
     private static const ROLE_SYSTEM_SPLITBUTTON:uint = 0x3e;

    /**
     * The object represents a button that drops down a list of items.
     */
     private static const ROLE_SYSTEM_BUTTONDROPDOWN:uint = 0x38;
...
```

System roles and states are predefined for all components in MSAA. The hexidecimal values for the constants can be found in the *ActionScript 3.0 Reference for the Adobe Flash Platform* under AccessibilityImplementation Constants. The MSAA documentation has a list of guidelines for accessible object implementations that documents which constants are used by which component types. Unfortunately, the User Interface Element Reference for MSAA omits the pop-up button. To figure out the appropriate role and state constants for the PopUpButton Accessibility Implementation, use the AccExplorer, Inspect32, and AccEvent tools that are available in the Microsoft Active Accessibility 2.0 SDK to examine a similar existing component in Windows. Fortunately, Flash Builder and the Eclipse plug-in use the pop-up button for a number of items in the main toolbar.

Launch Inspect32. Move your mouse over the Run icon button in the Flash Builder main toolbar. The Inspect32 tool updates to reveal the following information:

```
How found:Mouse move (199,85)
    hwnd=0x0006038A 32bit class="ToolbarWindow32" style=0x5601A945 ex=0x100000
Info:IAcc = 0x001F7A6C VarChild:[VT_I4=0x0]
Interfaces:IEnumVARIANT IOleWindow IAccIdentity
Impl:Remote oleacc proxy
Annotation ID:0100008000000000FCFFFFFF00000000
Name:"Run PopUpMenuButton"
Value:none [false]
Role:split button
State:hot tracked
Location:{l:192, t:72, w:38, h:22}
Description:none [false]
Kbshortcut:none [false]
DefAction:"Open"
Parent:none [false]:tool bar
Help:none [false]
Help Topic:none [false]
ChildCount:1
Window:0x0006038A class="ToolbarWindow32" style=0x5601A945 ex=0x100000
Children:"Open" : drop down button : hot tracked
Selection:none [empty]
Ancestors:none [false] : tool bar : normal
    none [false] : window : normal
    none [false] : client : default,focusable
    none [false] : window : normal
    none [false] : client : normal
    none [false] : window : normal
    none [false] : client : normal
    none [false] : window : normal
    "Flex Development - file:/C:/Program Files/Adobe/Flash Builder
4/plugins/com.adobe.flexbuilder.ui_4.0.nnnnnn/welcome/welcome.html - Adobe Flash Builder" :
client : normal"Flex Development - file:/C:/Program Files/Adobe/Flash Builder
4/plugins/com.adobe.flexbuilder.ui_4.0.nnnnnn/welcome/welcome.html - Adobe Flash Builder" :
window : sizeable,moveable,focusable"Desktop" : client : normal"Desktop" : window : normal
    [ No Parent ]
```

The Inspect32 tool reveals that the pop-up button should have the role `split button` and one child named `Open` with the role `drop down button`. When you mouse over the control or give it keyboard focus, the control’s state is `hot tracked`. In Flash terminology, “hot tracked” is a mouseover, or the button “over” state. A search on http://msdn.microsoft.com/ for "SplitButton accessibility" returns a more detailed and useful description of the SplitButton control's accessibiliy implementation: Working with Active Accessibility in the 2007 Office Fluent User Interface: The SplitButton Control. This more detailed description states that we should also expose the state of the child drop-down button that opens a popup menu as `has popup`.

Update the PopUpButtonAccImpl constructor function at line 199 to read:

```
...
    //----------------------------------------------------------------------- ---
-
    //
    //  Constructor
    //
    //------------------------------------------------------------------------ - ----
    /**
     * Creates a new PopUpButtonAccImpl instance for the specified PopUpButton component    .
     *
     * <p>Direct calls to the AccImpl subclass constructors are unneccessary.
     * When a Flex project is compiled with Generate accessible SWF file set to
     * <code>true</code>,
     * the compiler instantiates the accessibility implementations for the components
     * used in the project that have them by calling the static
     * <code>enableAccessibility()</code> method. </p   >
     *
     * @param master The UIComponent instance that this PopUpButtonAccImpl instance
     * is making accessible.
     *
     * @see ../../../html/help.html?content=accessible_4.html Configuring Flex
     * applications for accessibility
     */
    public function PopUpButtonAccImpl(master:UIComponent)
    {
        super(master);
        role = ROLE_SYSTEM_SPLITBUTTON;
    }
...
```

The constructor function defines the `role` property for the PopUpButtonAccImpl as `ROLE_SYSTEM_SPLITBUTTON` (0x3e). Update the `eventsToHandle()` method at line 237 to include any additional events to which the PopUpButtonAccImpl should listen from its master component. In this example, the events should be the same as for the ButtonAccImpl, `"click"` and `"labelChanged"`.

```
...
    //----------------------------------
    //  eventsToHandle
    //----------------------------------
    /**
     *  @inheritDoc
     */
    override protected function get eventsToHandle():Array
    {
        return super.eventsToHandle.concat([ "click", "labelChanged" ]);
    }
...
```

Update the `get_accRole()` method at line 247 to read as follows:

```
...
    /**
     * IAccessible method for returning system role for the component.
     * System roles are predefined for all the components in MSAA.
     * <p>A PopUpButton component, ( <code>childID == 0</code> ), reports
     * the role <code>ROLE_SYSTEM_SPLITBUTTON</code> (0x3E).
     * The PopUpButton's drop down arrow ( <code>childID == 1</code> )
     * reports the role <code>ROLE_SYSTEM_BUTTONDROPDOWN</code> (0x3B).</p>
     *
     * @param childID An unsigned integer corresponding to one of the component's child
     * elements as defined by
     * <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     * @return Role associated with the component.
     * @tiptext Returns the system role for the component
     * @helpid 3009
     *
     * @see ../../flash/accessibility/constants.html#roles AccessibilityImplementation
Constants: Object Roles
     * @see http://msdn.microsoft.com/en-us/library/ms696113(VS.85).aspx Microsoft
Accessibility Developer Center: IAccessible::get_accRole
     */
    override public function get_accRole(childID:uint):uint
    {
        if (childID == 0)
            return role;

        return ROLE_SYSTEM_BUTTONDROPDOWN;
    }
...
```

The `get_accRole()` method should return the appropriate role constant for the master component or for one of its children. The PopUpButtonAccImpl returns `ROLE_SYSTEM_SPLITBUTTON` (0x3e), while its child drop-down button returns `ROLE_SYSTEM_BUTTONDROPDOWN` (0x38).Add the following `get_accValue()` method override at line 272:

```
...
    /**
     * IAccessible method for returning the value of the PopUpButton
     * which is spoken out by the screen reader.
     * The PopUpButton should return value that would be applied when clicked.
     * For example, the value of the PopUpButton's label property.
     *
     * @param childID An unsigned integer corresponding to one of the component's
     * child elements as defined by
     * <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     *  @return Value String
     *
     *  @see http://msdn.microsoft.com/en-us/library/ms697312(VS.85).aspx Microsoft
Accessibility Developer Center: IAccessible::get_accValue
     *  @see http://msdn.microsoft.com/en-
us/library/bb404170.aspx#ActiveAccessibility2007OfficeFluentUI_TheSplitButtonControl Working
with Active Accessibility in the 2007 Office Fluent User Interface: The SplitButton Control
     */
    override public function get_accValue(childID:uint):String
    {
        var accValue:String;
        // local reference to the master component as a PopUpButton
        var popUpButton:PopUpButton = PopUpButton(master);

        if(childID == 0){
        // the PopUpButton component itself

            // local reference to the label
            var label:String = popUpButton.label;

            if(popUpButton.popUp
                && popUpButton.popUp is Menu
                && label != null
                && label != ""){
                // If the popUp exists and is a Menu and
                // the label exists and is not an empty string...

                var popUpMenu:Menu = popUpButton.popUp as Menu;
                if(popUpMenu.itemToLabel(popUpMenu.selectedItem) == label){
                // If the label matches the popUp menu's selectedIndex,
                // return the label as the accValue.
                    accValue = label;
                    if(popUpButton.accessibilityProperties &&
popUpButton.accessibilityProperties.shortcut){
                        // If a keyboard shortcut is defined in the
                        // PopUpButton's .accessibilityProperties object,
                        // append it to the returned accValue.
                        accValue + = "("+popUpButton.accessibilityProperties.shortcut+")";
                    }
                }

            }
        }

        return accValue;
    }
...
```

The `get_accValue()` method returns the appropriate value for the component or for one of its child elements. If the PopUpButton component's `label` property is equal to the label of the `selectedItem` in its popUp menu, the `get_accValue()` method should return the value of the `label` property. Update the `get_accState()` method at line 321 to read:

```
...
    /**
     * IAccessible method for returning the state of the PopUpButton.
     * States are predefined for all the components in MSAA.
     * Values are assigned to each state.
     * Depending upon the PopUpButton being pressed or released, its popUp being present
     * and opened or closed, and the PopUpButton being hovered over or focused,
     * a value is returned.
     *
     * @param childID An unsigned integer corresponding to one of the component's
     * child elements as defined by
     * <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     * @return State indicating whether the PopUpButton is pressed or released.
     *
     * @see #getChildIDArray()
     * @see flash.accessibility.AccessibilityImplementation#get_accState()
     * @see ../../flash/accessibility/constants.html#states AccessibilityImplementation
Constants: Object State Constants
     * @see http://msdn.microsoft.com/en-us/library/ms696191(VS.85).aspx Microsoft
Accessibility Developer Center: IAccessible::get_accState
     */
     override public function get_accState(childID:uint):uint
    {
        // the normal default state
        var accState:uint = 0;

        // local reference to the master component as a PopUpButton
        var popUpButton:PopUpButton = PopUpButton(master);

        if(childID == 1){
        // the drop-down button

            if(popUpButton.popUp){
            // if the PopUpButton's popUp property is defined,
            // indicate that the drop-down button has a pop up.
                accState = STATE_SYSTEM_HASPOPUP;
            }


            if(popUpButton.mx_internal::isShowingPopUp == true){
            // if the popUp is showing,
            // indicate that the drop-down button is pressed
                accState |= STATE_SYSTEM_PRESSED;
            }

        } else {
        // the PopUpButton component itself

            // the component state inherited from AccImpl
            // unavailable, normal, focusable, or focused
            accState = getState(childID);

            if (popUpButton.selected) { // if the popUpButton is selected,
            // indicate that popUpButton is pressed
```

```
                accState |= STATE_SYSTEM_PRESSED;
            }
        }

        var mouseX:Number = master.mouseX;
        var mouseY:Number = master.mouseY;
        var bounds:Rectangle = master.getBounds(master);
        if((mouseX >= bounds.x
            && mouseX <= (bounds.x + bounds.width)
            && mouseY >= bounds.y
            && mouseY <= (bounds.y + bounds.height))
            || (popUpButton.focusManager.getFocus() == popUpButton)){
            // if the parent popUpButton component or child drop-down button
            // has either mouse or keyboard focus,
            // indicate that it is hot-tracked.
            accState |= STATE_SYSTEM_HOTTRACKED;
        }
        return accState;
    }
...
```

The `get_accState() method` should return the appropriate state constant or state constants for the master component or for one of its children. If the PopUpButton has a `popUp` property, its child drop-down button should have the state `STATE_SYSTEM_HASPOPUP` (0x40000000). If the popUp is showing, the drop-down button should also have the state `STATE_SYSTEM_PRESSED` (0x00000008). If the PopUpButton itself is selected, it should have the state `STATE_SYSTEM_PRESSED` (0x00000008). Both the PopUpButton and its child drop-down button should have the state `STATE_SYSTEM_HOTTRACKED` (0x00000080) when a user mouses over the PopUpButton or when it has keyboard focus.Add the following `getChildIDArray()` method override at line 398:

```
...
    /**
     * Method to return an array of childIDs.
     *
     * @return An array of unsigned integer IDs with a length of one
     * for the drop-down menu button.
     */
    override public function getChildIDArray():Array
    {
        var childIDs:Array = [1];

        return childIDs;
    }
...
```

The `getChildIDArray`() method returns an array containing the unsigned integer IDs of all child elements in the accessibility implementation. The length of the array may be zero. The IDs in the array should appear in the same logical order as the child elements they represent. This method is mandatory for any accessibility implementation that can contain child elements; otherwise, do not implement it. For the PopUpButtonAccImpl, the `getChildIDArray()` method returns an array containing a single element with a value of 1, which represents the PopUpButton's child drop-down button element.Update the `get_accDefaultAction()` method at line 410 to return the appropriate default action for the master component or for one of its children. For the PopUpButton component itself, the method returns “Press”. For the component's child drop-down button, the method returns “Open.”

```
...
    /**
     * IAccessible method for returning the default action of the PopUpButton,
     * which is Press for the component itself and Open for its child drop-down
     * menu button.
     *
     * @param childID An unsigned integer corresponding to one of the component's
     * child elements as defined by
     *  <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     *  @return DefaultAction String
     *
     *  @see http://msdn.microsoft.com/en-us/library/ms696144(VS.85).aspx Microsoft
Accessibility Developer Center: IAccessible::get_accDefaultAction
     */
    override public function get_accDefaultAction(childID:uint):String
    {
        if(childID == 0){
            return "Press";
        }
        return "Open";
    }
...
```

Update the `addDoDefaultAction()` method at line 439 to read as follows:

```
...
    /**
     * IAccessible method for performing the default action of the PopUpButton, which is Press
for the component itself and Open for its child drop-down menu button.
     *
     * @param childID An unsigned integer corresponding to one of the component's
     * child elements as defined by
     *  <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     *  @see http://msdn.microsoft.com/en-us/library/ms696119(VS.85).aspx Microsoft
Accessibility Developer Center: IAccessible::accDoDefaultAction
     */
    override public function accDoDefaultAction(childID:uint):void
    {
        var popUpButton:PopUpButton = master as PopUpButton;
        if (childID==0)
        {
            var event:KeyboardEvent = new KeyboardEvent(KeyboardEvent.KEY_DOWN);
            event.keyCode = Keyboard.SPACE;
            master.dispatchEvent(event);

            event = new KeyboardEvent(KeyboardEvent.KEY_UP);
            event.keyCode = Keyboard.SPACE;
            master.dispatchEvent(event);

        } else if(childID == 1){

            if(popUpButton.mx_internal::isShowingPopUp == true){
                popUpButton.close();
            } else {
                popUpButton.open();
            }
        }
    }
...
```

The `accDoDefaultAction()` method executes the default action for the master component or for one of its children. For the PopUpButton component itself, the method simulates a button click. For the child drop-down button element, the method either opens or closes the popUp. Update the getName() method at line 463 to read as follows:

```
...
    /**
     * Method for returning the name of the PopUpButton which is spoken out by
     * the screen reader.
     *  The PopUpButton should return the label inside as the name of the PopUpButton.
     *  The child drop-down button should return "Open" or "Close".
     *
     * @param childID An unsigned integer corresponding to one of the component's
     * child elements as defined by
     *  <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     * @return Name string corresponding to the PopUpButton label.
     * "Open" or "Close" for the child drop-down button.
     */
    override protected function getName(childID:uint):String
    {
        var popUpButton:PopUpButton = master as PopUpButton;
        var popUp:UIComponent = popUpButton.popUp as UIComponent;

        if(popUp && popUp is Menu){
        // a tweak to add Context as the name of the popUp menu if no other name is defined
            if(!popUp.accessibilityProperties){
                popUp.accessibilityProperties = new AccessibilityProperties();
            }
            if(popUp is Menu && popUp.accessibilityProperties.name == ""){
                popUp.accessibilityProperties.name = "Context";
            }
        }
        if(childID == 1){
        // For the drop-down button, if the popUp is showing, return Close,
        // otherwise return Open.
            return (popUpButton.mx_internal::isShowingPopUp == true) ? "Close" : "Open";
        }

        // For the popUpButton component itself, return the label property.
        var label:String = popUpButton.label;
        return label != null && label != "" ? label : "";
    }
...
```

The `getName()` method returns the appropriate name for the master component or for one of its children. For the PopUpButton component itself, the method should return its label property as its name. The component's child drop-down button returns either “Open” or “Close”, depending on whether or not the popUp is showing. For the PopUpButton, this method also adds “Context” as the name of the popUp menu if no other name has been defined.Add the following `accLocation()` method at line 498:

```
...
    /**
     * IAccessible method for returning the bounding box of a the PopUpButton
     * or its child drop-down arrow.
     *
     * @param childID An unsigned integer corresponding to one of the component's
     * child elements as defined by
     *  <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     *  @return The PopUpButton or its drop-down button.
     *
     *  @see http://msdn.microsoft.com/en-us/library/ms696118(VS.85).aspx Microsoft
Accessibility Developer Center: IAccessible::accLocation
     */
    override public function accLocation(childID:uint):*
    {

        // the master component
        var location:* = master;

        if(childID == 1){
        // calculate the rectangle location of the child drop-down button
            var popUpButton:PopUpButton = master as PopUpButton;
            var popUpButtonRect:Rectangle = popUpButton.getRect(popUpButton);
            var arrowButtonsWidth:Number =
popUpButton.mx_internal::getArrowButtonsWidth();
            location = new Rectangle(
                                popUpButtonRect.x + popUpButton.width -
arrowButtonsWidth,
                                popUpButtonRect.y,
                                arrowButtonsWidth,
                                popUpButton.height
                            );
        }

        return location;

    }
...
```

The `accLocation()` method returns the bounding box of the master component, or that of one of its children, relative to the master component's stage. The PopUpButton returns itself as the its bounding box location. Its child drop-down button returns its bounding box as a Rectangle with its bounds determined programmatically.Update the `eventHandler()` method at line 535 to handle any events to which the PopUpButtonAccImpl should listen from its master component. In this example, the handled events should be the same as for the ButtonAccImpl, `"click"` and `"labelChanged"`.

```
...
    /**
     * Override the generic event handler.
     * Each AccImpl subclass must implement this method
     * to listen for events from its master component.
     *
     *  @param event The event object.
     *
     *  @see mx.accessibility.AccImpl#eventHandler()
     */
    override protected function eventHandler(event:Event):void
    {
        switch (event.type)
        {
            case "click":
            {
                Accessibility.sendEvent(master, 0, EVENT_OBJECT_STATECHANGE);
                Accessibility.updateProperties();
                break;
            }

            case "labelChanged":
            {
                Accessibility.sendEvent(master, 0, EVENT_OBJECT_NAMECHANGE);
                Accessibility.updateProperties();
                break;
            }

        }
    }
...
```

The `eventHandler()` method handles events from the master component and sends the appropriate object or system event to MSAA using the `flash.accessibility.Accessibility.sendEvent()` method. Depending on the object or system event it receives, the MSAA system will call the IAccessible method to retrieve the appropriate value from the object that sent the event.

Testing the accessibility implementation

Once you've added the accessibility mix-in methods to your component and have created the AccImpl subclass, you can test the accessibility implementation.Copy the following MXML code sample into the default MXML application for your PopUpMenuButton Flex project. The application is a slightly modified version of the PopUpMenuButton example described in the article PopUpMenuButton control. The modification adds event handlers to retain the `selectedItem` in the popUp menu when the menu is opened and closed, which allows you to test the `PopUpButtonAccImpl.get_accValue()` method.

```
<?xml version="1.0"?>
<!-- m  enus/Pop U pMenuButton Control.mxml     - - >
 <s:Application xm lns :fx = "http://ns.adobe.co m/mx ml/2009"
    xmlns:s="library: //ns. adobe.com/flex/spark"
    xmlns:mx=" librar y:// n s.adob e .com/flex/mx"
    viewSourceURL="srcview/index.html">

    <fx:Script>
        <![CDATA [
            import mx.events.DropdownEvent;
            impor t  mx.events.MenuEvent;
            import mx.controls.Menu;

            private var selectedIndex:int = 2;

            // The initData function sets the initial value of the button
            // label by setting the Menu subcontrol's selectedIndex property.
            // You must cast the popUp property to a Menu.
            private function initData():void {
                var menu:Menu = Menu(pb2.popUp);
                menu.selectedIndex = selectedIndex;
                pb2.addEventListener(MenuEvent.ITEM_CLICK, itemClickHandler);
                pb2.addEventListener(DropdownEvent.OPEN, openHandler);
                pb2.addEventListener(DropdownEvent.CLOSE, closeHandler);
            }

            private function itemClickHandler(event:MenuEvent):void {
            var menu:Menu = Menu(event.currentTarget.popUp);
            selectedIndex=menu.selectedIndex;
            }

            private function openHandler(event:DropdownEvent):void {
                var menu:Menu = Menu(pb2.popUp);
                menu.selectedIndex = selectedIndex;
            }

            private function closeHandler(event:DropdownEvent):void {
                var menu:Menu = Menu(pb2.popUp);
                selectedIndex = (selectedIndex != menu.selectedIndex) ?
                menu.selectedIndex : selectedIndex;
                menu.selectedIndex = selectedIndex;
            }
        ]]>
    </fx:Script>
    <fx:XML format="e4x" id="dp2">
        <root>
            <editItem label="Cut"/>
            <editItem label="Copy"/>
            <editItem label="Paste"/>
            <separator type="separator"/>
            <editItem label="Delete"/>
        </root>
    </fx:XML>
    <mx:PopUpMenuButton id="pb2"
        dataProvider="{dp2}"
        labelField="@label"
        showRoot="false"
        creationComplete="initData();" />
</s:Application>
```

Compile and debug the application in either Internet Explorer or Firefox. Open the MSAA SDK Tool Inspect32. Focus on the Flash movie in the browser window and navigate to focus on the PopUpMenu instance in your application using the the keyboard or the mouse. Inspect32 should track your focus and update its display window with information similar to the following:

```
How found:Focus [o:0xFFFFFFFC,c:0x1]
    hwnd=0x000C0976 32bit class="MacromediaFlashPlayerActiveX" style=0x56000000 ex=0x0
Info:IAcc = 0x001F1568 VarChild:[VT_I4=0x0]
Interfaces:IEnumVARIANT
Impl:Remote native IAccessible
Annotation ID:[not supported]
Name:"Paste"
Value:"Paste"
Role:split button
State:focused,hot tracked,focusable
Location:{l:802, t:147, w:63, h:22}
Description:none [mnfd]
Kbshortcut:none [mnfd]
DefAction:"Press"
Parent:none [mnfd]:client
Help:none [mnfd]
Help Topic:none [mnfd]
ChildCount:1
Window:0x000C0976 class="MacromediaFlashPlayerActiveX" style=0x56000000 ex=0x0
Children:"Open" : drop down button : hot tracked,has popup
Selection:none [mnfd]
Ancestors:none [mnfd] : client : focusable
    [ No Parent ]
```

Inspect32 reveals how the PopUpButton Accessibility Implementation exposes itself to MSAA, as an object with the role “split button,”, with name and value equal to “Paste,” having the default action “Press.”

The object's state includes “focused” and “hot tracked”. It contains a single child with the role “drop down button,” that has the name “Open.” The child object's state includes “hot tracked” and “has popup.” Inspect32 also allows you to test navigation and interaction programmatically.

For example, with the application in focus, type Ctrl+Shift+F12 to navigate to the next item or type Ctrl+Shift+F11 to navigate to the previous item in the MSAA hierarchy. Use Ctrl+Shift+F12 and/or Ctrl+Shift+F11 to set focus on the PopUpMenuButton instance's child drop down button. Type Ctrl+Shift+F2 to execute the default action for the drop down button. This action should open the popUp menu. The drop down button name should change to “Close,” and its state should indicate “pressed.” Type Ctrl+Shift+F2 again to close the menu.

Once you're fairly confident that your accessibility implementation is returning appropriate values for its IAccessible methods, you should test again using a screen reader that is capable of interpreting Flash applications, such as Freedom Scientific's JAWS or GW Micro's Window-Eyes.

It is not uncommon for a component to behave unexpectedly when testing it with a screen reader. For example, the keyboard command Ctrl+Down arrow, which is used in the PopUpButton component to open the popUp menu, is also the standard keyboard command in JAWS to move one paragraph down in the text.

Using Ctrl+Shift+Down arrow instead of Ctrl+Down arrow will open the menu, but it's not obvious to the user. A better solution is to update the PopUpButton component, to allow it to open by pressing the Down arrow.Update the `PopUpButton.keyDownHandler()` method at line 907 in the PopUpButton class file for the PopUpMenuButton Flex project to read:

```
...
    override protected function keyDownHandler(event:KeyboardEvent):void
    {
        super.keyDownHandler(event);

        // listen for both Ctrl+Down Arrow and Down Arrow events
        // if the popUp menu is not already showing
        if ((event.ctrlKey && event.keyCode == Keyboard.DOWN)
        || (event.keyCode == Keyboard.DOWN &&!showingPopUp))
        {
            openWithEvent(event);
            event.stopPropagation();
        }
        else if ((event.ctrlKey && event.keyCode == Keyboard.UP) ||
                (event.keyCode == Keyboard.ESCAPE))
        {
            closeWithEvent(event);
            event.stopPropagation();
        }
        else if (event.keyCode == Keyboard.ENTER &&mp; showingPopUp)
        {
            // Redispatch the event to the popup
            // and let its keyDownHandler() handle it.
            _popUp.dispatchEvent(event);
            closeWithEvent(event);
            event.stopPropagation();
        }
        else if (showingPopUp &&
            (event.keyCode == Keyboard.UP ||
            event.keyCode == Keyboard.DOWN ||
            event.keyCode == Keyboard.LEFT ||
            event.keyCode == Keyboard.RIGHT ||
            event.keyCode == Keyboard.PAGE_UP ||
            event.keyCode == Keyboard.PAGE_DOWN))
        {
            // Redispatch the event to the popup
            // and let its keyDownHandler() handle it.
            _popUp.dispatchEvent(event);
            event.stopPropagation();
        }
    }
...
```

Another peculiarity of JAWS is that it announces the role of the PopUpMenuButton component as a “button” as opposed to a “split button.” Even if you are providing a correct accessibility implementation to MSAA, if it is of a system role that does not have a precedent in Flex or Flash, like the PopUpButton, the screen reader may not have been scripted to understand how to interpret an object with that system role in a Flash application. The short term solution is to add the system role to the name returned by the `PopUpButtonAccImpl.getName()` method.

Update the `getName()` method at line 463 in the PopUpButtonAccImpl class definition to read:

```
...
    /**
     * Method for returning the name of the PopUpButton that is spoken
     * by the screen reader.
     *  The PopUpButton should return the label inside as the name of the PopUpButton.
     *  The child drop-down button should return "Open" or "Close".
     *
     *  @param childID An unsigned integer corresponding to one of the component's
     * child elements as defined by
     *  <code><a href="#getChildIDArray()">getChildIDArray()</a></code>.
     *
     *  @return Name string corresponding to the PopUpButton label.
     * "Open" or "Close" for the child drop-down button.
     */
    override protected function getName(childID:uint):String
    {
        var popUpButton:PopUpButton = master as PopUpButton;
        var popUp:UIComponent = popUpButton.popUp as UIComponent;

        if(popUp && popUp is Menu){
```

```
        // a tweak to add Context as the name of the popUp menu if no other name is defined
            if(!popUp.accessibilityProperties){
                popUp.accessibilityProperties = new AccessibilityProperties();
            }
            if(popUp is Menu && popUp.accessibilityProperties.name == ""){
                popUp.accessibilityProperties.name = "Context";
            }
        }
        if(childID == 1){
        // For the drop-down button, if the popUp is showing, return Close,
        // otherwise return Open.
            return (popUpButton.mx_internal::isShowingPopUp == true) ? "Close" : "Open";
        }

        // For the popUpButton component itself, return the label property.
        var label:String = popUpButton.label;
        return label != null && label != "" ? label + " Split Button" : "Split Button";

    }
...
```

Accessibility for hearing-impaired users

To provide accessibility for hearing-impaired users, you can include captions for audio content that is integral to the comprehension of the material presented. A video of a speech, for example, would probably require captions for accessibility, but a quick sound associated with a button probably would not require a caption.

Testing accessible content

When you test your accessible applications, follow these recommendations:

- Ensure that you have enabled accessibility. For more information, see “Configuring Flex applications for accessibility” on page 2128.
- Test and verify that users can navigate your interactive content effectively by using only the keyboard. This can be an especially challenging requirement, because different screen readers work in different ways when processing input from the keyboard, which means that your content might not receive keystrokes as you intended. Ensure that you test all keyboard shortcuts.
- Ensure that graphic content has alternative representation by checking for tool tips on every image.
- Ensure that your content is resizable or sized to work with screen magnifiers. It should clearly present all material at smaller sizes.
- Ensure that any substantive audio is captioned. To test for content loss, try using your application with your sound muted or wearing headphones that pipe in unrelated music.
- Ensure that you use a clean and simple interface to help screen readers and users with cognitive impairments navigate through your application. Give your application to people who have never seen it before and watch while they try to perform key tasks without any assistance or documentation. If they fail, consider modifying your interface.
- Test your application with a screen reader, or have it tested by users of screen readers.

Chapter 8: Developer tools

Building overview

About the Flex development tools

Configuration files

You must be familiar with the ways to configure your development environment. Adobe Flex Software Development Kit (SDK) primarily provide XML files that you use to configure the settings. The flex-config.xml file defines the default compiler options for the compilers.

In addition to server and compiler configuration settings, you can also modify the messaging and data management settings, the JVM heap size, Adobe® Flash® Player settings, and logging and caching settings.

For more information about configuring your Flex SDK environment, see "SDK configuration" on page 96.

Compilers

Flex includes application compilers and component compilers. You use the application compilers to compile SWF files from MXML and other source files. You use the component compilers to compile SWC files from component files. You can then use SWC files as dynamic or static libraries with your Flex applications.

The application compilers take the following forms:

- Adobe® Flash™ Builder™ project compiler. The Flash Builder application compiler is opened by Flash Builder for Flex Projects and ActionScript Projects.
- mxmlc command-line compiler. You launch the mxlmc compiler from the command line to create a SWF file that you then deploy to a website.
- fcsh compiler shell. Provides an optimized environment for using the mxmlc command-line compiler. For more information, see "Using fcsh, the Flex compiler shell" on page 2207.

The component compilers take the following forms:

- Flash Builder library project compiler. The Flash Builder component compiler is opened by Flash Builder for Flex library projects.
- compc command-line compiler. You open the compc compiler from the command line to create SWC files. You can use these SWC files as static component libraries, themes, or runtime shared libraries (RSLs).

For information on using the compilers, see "Flex compilers" on page 2164.

Debugger

To test your applications, you run the application SWF files in a web browser or the stand-alone Flash Player. If you encounter errors in your applications, you can use the debugging tools to set and manage breakpoints in your code; control application execution by suspending, resuming, and terminating the application; step into and over the code statements; select critical variables to watch; evaluate watch expressions while the application is running; and so on.

Flex provides the following debugging tools:

Flash Builder debugger The Flash Builder Debugging perspective provides all of the debugging tools you expect from a robust, full-featured development tool. You can set and manage breakpoints; control application execution by suspending, resuming, and terminating the application; step into and over the code; watch variables; evaluate expressions; and so on. For more information, see Debugging Tools in Flash Builder.

The fdb command-line debugger The fdb command-line debugger provides a command-line interface to the debugging experience. With fdb, you can step into code, add breakpoints, check variables, and perform many of the same tasks you can with the Flash Builder visual debugger. For more information, see "Command-line debugger" on page 2209.

AIR Debug Launcher (ADL) ADL is a command line debugger for Adobe® AIR™ applications that you can use outside of Flash Builder.

Loggers

You can log messages at several different points in a Flex application's life cycle. You can log messages when you compile the application, when you deploy it to a web application server, or when a client runs it. You can log messages on the server or on the client. These messages are useful for informational, diagnostic, and debugging activities.

Flex includes the following logging mechanisms that you use when working with Flex applications.

Client-side logging When you use the debugger version of Flash Player or start your AIR application using AIR Debug Launcher, you can use the trace() global method to write out messages or configure a TraceTarget to customize log levels of applications for data services-based applications. For more information, see "Client-side logging and debugging" on page 2227.

Compiler logging When compiling your Flex applications from the command line and in Flash Builder, you can view deprecation and warning messages, and sources of fatal errors. For more information, see "Compiler logging" on page 2237.

About application files

Flex applications can use many types of application files such as classes, component libraries, theme files, and Runtime Shared Libraries (RSLs).

Component classes

You can use any number of component classes in your Flex applications. These classes can be MXML or ActionScript files. You can use classes to extend existing components or define new ones.

Component classes can take the form of MXML, ActionScript files, or as SWC files. In MXML or ActionScript files, the components are not compiled but reside in a directory structure that is part of your compiler's source path. SWC files are described in "SWC files" on page 2162.

Component libraries are not dynamically linked unless they are used in a Runtime Shared Library (RSL). Component classes are statically linked at compile time, which means that they must be in the compiler's source path. For information about creating and using custom component classes, see *"Custom Flex components" on page 2356*.

Related links

SWC files

A *SWC* file is an archive file for Flex components and other assets. SWC files contain a SWF file and a catalog.xml file. The SWF file inside the SWC file implements the compiled component or group of components and includes embedded resources as symbols. Flex applications extract the SWF file from a SWC file, and use the SWF file's contents when the application refers to resources in that SWC file. The catalog.xml file lists of the contents of the component package and its individual components.

You compile SWC files by using the component compilers. These include the compc command-line compiler and the Flash Builder Library Project compiler. SWC files can be component libraries, RSLs, theme files, and resource bundles.

To include a SWC file in your application at compile time, it must be located in the library path. For more information about SWC files, see "About SWC files" on page 2205.

Component libraries

A component library is a SWC file that contains classes and other assets that your Flex application uses. The component library's file structure defines the package system that the components are in.

Typically, component libraries are statically linked into your application, which means that the compiler compiles it into the SWF file before the user downloads that file.

To build a component library SWC file, you use the `include-classes`, `include-namespaces`, and `include-sources` component compiler options. For more information on building component libraries, see "Using compc, the component compiler" on page 2194.

Runtime Shared Libraries

You can use shared assets that can be separately downloaded and cached on the client in Flex. These shared assets are loaded by multiple applications at run time, but must be transferred only once to the client. These shared files are known as *Runtime Shared Libraries* or *RSLs*.

RSLs are the only kind of application asset that is dynamically linked into your Flex application. When you compile your application, the RSL source files must be available to the compiler so that it can perform proper link checking. The assets themselves are not included in the application SWF file, but are only referenced at run time.

To create an RSL SWC file, you add files to a library by using the `include-classes` and `include-namespaces` component compiler options. To use RSLs when compiling your application, you use the `external-library-path`, `externs`, `load-externs`, and `runtime-shared-libraries` application compiler options. The `external-library-path`, `externs`, and `load-externs` options provide the compile-time location of the libraries. The `runtime-shared-libraries` option provides the run-time location of the shared library. The compiler requires this for dynamic linking.

For more information, see "Runtime Shared Libraries" on page 253.

Themes

A *theme* defines the look and feel of a Flex application. A theme can define something as simple as the color scheme or common font for an application, or it can be a complete reskinning of all the components used by the application.

Themes usually take the form of a SWC file. However, themes can also be composed of a CSS file and embedded graphical resources, such as symbols from a SWF file.

Theme files must be available to the compiler at compile-time. You build a theme file by using the `include-file` and `include-classes` component compiler options to add skin files and style sheets to a SWC file. You then reference the theme SWC file when you compile the main Flex application by using the `theme` application compiler option.

For more information about themes, see "About themes" on page 1561.

Resource bundles

You can package libraries of localized properties files and ActionScript classes into a SWC file. The application compiler can then statically use this SWC file as a resource bundle. For more information about creating and using resource bundles, see "Localization" on page 2055.

Other assets

Other application assets include images, fonts, movies, and sound files. You can embed these assets at compile time or access them at run time.

When you embed an asset, you compile it into your application's SWF file. The advantage of embedding an asset is that it is included in the SWF file, and can be accessed faster than when the application has to load it from a remote location at run time. The disadvantage of embedding an asset is that your SWF file is larger than if you load the resource at run time.

The alternative to embedding an asset is to load the asset at run time. You can load an asset from the local file system in which the SWF file runs, or you can access a remote asset, typically though an HTTP request over a network.

Embedded assets load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded assets also require you to recompile your applications whenever your asset files change.

For more information, see "Embedding assets" on page 1699.

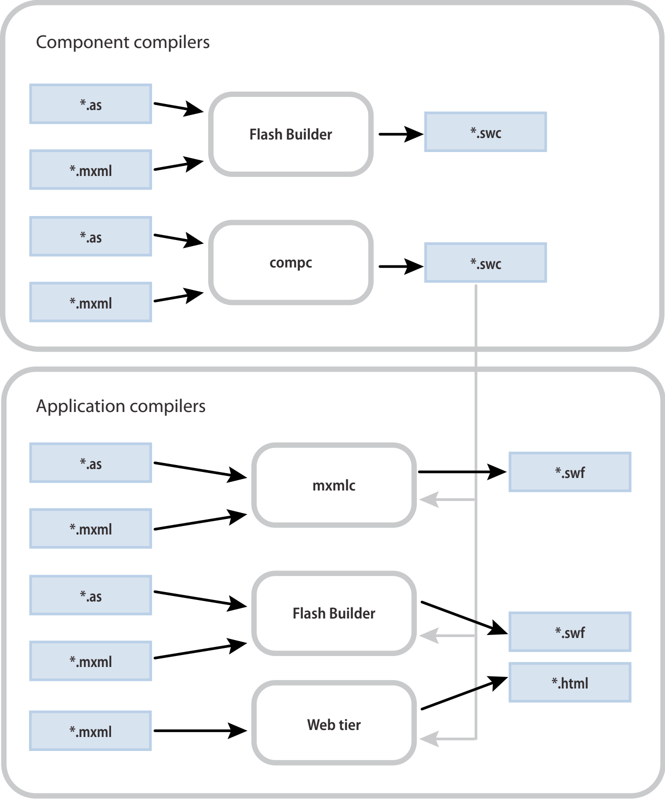
Flex compilers

About the Flex compilers

Flex includes the following application and component compilers:

- Application compilers. The application compilers create SWF files from MXML, ActionScript, and other assets such as images, SWF files, and SWC files.
- Component compilers. The component compilers create SWC files from the same kinds of files. The application compilers then use the SWC files as component libraries, themes, or RSLs.

The following example shows the input and output of the Flex compilers:



You open the application compiler with the mxmlc command-line tool or with the Flash Builder Build Project option. You open the component compiler with the Flash Builder Build Project option for a Library Project or with the compc command-line tool.

About the application compilers

The application compilers create SWF files that are run in an Adobe™ Flash® Player client or in an Adobe AIR™ application. The client can be a stand-alone Flash Player or a Flash Player in a browser, which takes the form of an ActiveX control for Microsoft Internet Explorer or a plug-in for Netscape-based browsers.

Flash Builder project compiler The Flash Builder application compiler is opened by Flash Builder for Flex Projects and ActionScript Projects. (The component compiler is used for Library Projects.) It is similar in functionality to the mxmlc command-line compiler, although the way you set options is different. You use this compiler to compile Flash Builder projects that you will later deploy. For more information, see "Using the Flash Builder application compiler" on page 2166.

The mxmlc command-line compiler You open the mxmlc compiler from the command line to create a SWF file that you then deploy to a website. Typically, you pass the name of the application's root MXML file to the compiler. The output is a SWF file. For more information, see "Using the mxmlc application compiler" on page 2166.

The Flash Builder compiler and mxmlc compiler have similar sets of options. These are described in "About the application compiler options" on page 2175.

You can compile applications that are written entirely in ActionScript and contain no MXML. You can compile these "ActionScript-only" applications with the Flash Builder and mxmlc compilers.

Using the Flash Builder application compiler

You use the Flash Builder application compiler to create SWF files from MXML, ActionScript, and other source files. You use this compiler to precompile SWF files that you deploy later.

To open the Flash Builder application compiler, you select Project > Build. The Flash Builder application compiler is opened by Flash Builder for Flex Projects and ActionScript Projects. (You use the component compiler for Library Projects.)

To edit the compiler settings, use the settings on the Project > Properties > Flex Compiler dialog box. For information on the compiler options, see "About the application compiler options" on page 2175.

The Flash Builder compiler has the same options as the mxmlc compiler. Some options are implemented with GUI controls in the Flex Compiler dialog box. To set the source path and library options, select Project > Properties > Flex Build Path and use the Flex Build Path dialog box.

You can set the values of most options in the Additional Compiler Arguments field by using the same syntax as on the command line. For information about the command-line syntax, see "About the command-line compilers" on page 2169.

In the Additional Compiler Arguments field, you can substitute a path to the SDK directory by using the `${flexlib}` token, as the following example shows:

```
-include-libraries "${flexlib}/libs/automation.swc" "${flexlib}/libs/automation_agent.swc"
```

By default, Flash Builder exposes the compilation options through the project properties. If you want to use a configuration file, you can create your own and pass it to the compiler by using the `load-config` option. For more information on setting compiler options with configuration files, see "About configuration files" on page 2171.

In addition to generating SWF files, the Flash Builder compiler also generates an HTML wrapper that you can use when you deploy the new application. The HTML wrapper includes the `<object>` and `<embed>` tags that reference the new SWF file, as well as scripts that support history management and player version detection. For more information about the HTML wrapper, see "Creating a wrapper" on page 2552.

The Flash Builder application compiler uses incremental compilation by default. For more information on incremental compilation, see "About incremental compilation" on page 2191.

Using the mxmlc application compiler

You use the mxmlc command-line compiler to create SWF files from MXML, AS, and other source files. You can open it as a shell script and executable file for use on Windows and UNIX systems. You use this compiler to precompile applications that you deploy later.

The command-line compiler is installed with Flex SDK. It is in the *flex_install_dir*/bin directory in Flex SDK. The compiler is also included in the default Flash Builder installation, in the *flash_builder_install_dir*/sdks/*sdk_version*/bin directory.

To use the mxmlc utility, you should understand its syntax and how to use configuration files. For more information, see "About the command-line compilers" on page 2169.

The basic syntax of the mxmlc utility is as follows:

```
mxmlc [options] target_file
```

The default option is the target file to compile into a SWF file, and it is required to have a value. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file; for example:

```
mxmlc -option arg1 arg2 arg3 -- target_file.mxml
```

To see a list of options for mxmlc, you can use the `helplist` option, as the following example shows:

```
mxmlc -help list
```

To see a list of all options available for mxmlc, including advanced options, you use the following command:

```
mxmlc -help list advanced
```

The default output of mxmlc is *filename*.swf, where *filename* is the name of the root application file. The default output location is in the same directory as the target, unless you specify an output file and location with the `output` option.

The mxmlc command-line compiler does not generate an HTML wrapper. You must create your own wrapper to deploy a SWF file that the mxmlc compiler produced. The wrapper is used to embed the SWF object in the HTML tag. It includes the `<object>` and `<embed>` tags, as well as scripts that support Flash Player version detection and history management. For information about creating an HTML wrapper, see "Creating a wrapper" on page 2552.

The mxmlc utility uses the default compiler settings in the flex-config.xml file. This file is in the *flex_sdk_dir*/frameworks/ directory. You can change the settings in this file or use another custom configuration file. For more information on using configuration files, see "About configuration files" on page 2171.

The mxmlc compiler is highly customizable with a large set of options. For information on the compiler options, see "About the application compiler options" on page 2175.

You can also open the mxmlc compiler with the `java` command on the command line. For more information, see "Invoking the command-line compilers with Java" on page 2170.

About the component compiler

You use the component compiler to generate a SWC file from component source files and other asset files such as images and style sheets. A SWC file is an archive of components and other assets. For more information about SWC files, see "About SWC files" on page 2205.

In some ways, the component compiler is similar to the application compiler. The application compiler produces a SWF file from one or more MXML and ActionScript files; the component compiler produces a SWC file from its input files. SWC files are compressed files that contain a SWF file (library.swf), asset files, and a catalog.xml file.

You use the component compiler to create the following kinds of assets:

Component libraries Component libraries are SWC files that contain one or more components that are used by applications. SWC files also contain the namespace information that describe the contents of the SWC file. For more information about component libraries, see "About SWC files" on page 2205.

Runtime shared libraries (RSLs) RSLs are external shared assets that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time. For more information on RSLs, see "Runtime Shared Libraries" on page 253.

Themes Themes are a combination of skins and Cascading Style Sheets (CSS). You use themes to define the look and feel of an application built with Flex. For more information on themes, see "Styles and themes" on page 1492.

You use the component compiler in the following ways:

Flash Builder Library Project compiler Flash Builder uses the component compiler when you create a Library Project. (The application compiler is used for Flex Projects and ActionScript Projects). For more information, see "Using the Flash Builder component compiler" on page 2168.

The compc command-line compiler You open the compc compiler from the command line to create a SWC file. For more information, see "Using the compc component compiler" on page 2168.

The component compiler has many of the same options as the application compilers, as described in "About the application compiler options" on page 2175. The component compiler has additional options as described in "About the component compiler options" on page 2194.

Like the application compiler, you can use the `load-config` option to point the component compiler to a configuration file, rather than specify command-line options or set the options in the Flex Compiler dialog box.

When you compile a SWC file, store the new file in a location that is not the same as the source files. If both sets of files are accessible by Flex when you compile your application, unexpected behavior can occur.

The SWC files that are produced by the component compilers do not require an HTML wrapper because they are used as themes, RSLs, or component libraries that are input to other compilers to create applications. You never deploy a SWC file that users can request directly.

Using the Flash Builder component compiler

You use the component compiler in Flash Builder to create SWC files. You do this by setting up a Flex Library Project using the New Library Project command. You add MXML and ActionScript components, style sheets, SWF files, and other assets to the project.

To open the Flash Builder component compiler, select Project > Build Project for your Flex Library Project.

You edit the compiler settings by using the settings on the Project > Properties > Flex Compiler dialog box. Using the Additional Compiler Arguments field in this dialog box, you can set compiler options as if you were using the command-line compiler. For information about the syntax for setting options in the Flex Compiler dialog box, see "About the command-line compilers" on page 2169.

You can set the value of some options using the GUI controls. To set the resource options in the Flex Library Build Path dialog box, select Project > Properties > Flex Library Build Path.

Flash Builder does not expose a default configuration file to set compiler options but you can create your own and pass it to the compiler with the `load-config` option. For more information on setting compiler options with configuration files, see "About configuration files" on page 2171.

Using the compc component compiler

You use the compc command-line compiler to compile SWC files from MXML, ActionScript, and other source files such as style sheets, images, and SWF files.

You can open the compc compiler as a shell script and executable file for use on Windows and UNIX systems. It is in the *flex_install_dir*/bin directory in Flex SDK

To use the compc compiler, you should understand how to pass options and use configuration files. For more information, see "About the command-line compilers" on page 2169.

The syntax of the compc compiler is as follows:

```
compc [options] -include-classes class [...]
```

The default option for compc is `include-classes`. At least one of the "include-" options is required.

To see a list of supported options for compc, you can use the `helplist` option, as the following example shows:

```
compc -help list
```

The compc compiler uses the default compiler settings in the flex-config.xml file. Like the application compiler, you can change these settings or use another custom configuration file. Unlike the application compiler, however, the component compiler does not support the use of default configuration files.

You cannot use the compc compiler to create a SWC file from a FLA file or other file created in the Adobe® Flash® authoring environment.

You can also open the compc compiler with the `java` command on the command line. For more information, see "Invoking the command-line compilers with Java" on page 2170.

About the command-line compilers

You use the mxmlc and compc command-line compilers to compile your MXML and AS files into SWF and SWC files. You can use the utilities to precompile applications that you want to deploy on another server or to automate compilation in a testing environment.

To use the command-line compilers, you must have a Java run-time environment in your system path.

For Flex SDK, the command-line compilers are located in the *flex_install_dir*/bin directory. For Flash Builder, the compilers are located in the *flash_builder_install_dir*/sdks/*sdk_version*/bin directory.

When using mxmlc and compc on the command line, you can also use a configuration file to store your options rather than list them on the command line. You can store command-line options as XML blocks in a configuration file. For more information, see "About configuration files" on page 2171.

Command-line syntax

The mxmlc and compc compilers take many options. The options are listed in the help which you can view with the `help` option, as the following example shows:

```
mxmlc -help
```

This displays a menu of choices for getting help. The most common choice is to list the basic configuration options:

```
mxmlc -help list
```

To see advanced options, use the `list advanced` option, as the following example shows:

```
mxmlc -help list advanced
```

To see a list of entries whose names or descriptions include a particular String, use the following syntax:

```
mxmlc -help pattern
```

The following example returns descriptions for the `external-library-path`, `library-path`, and `runtime-shared-libraries` options:

```
mxmlc -help list library
```

For a complete description of mxmlc options, see "About the application compiler options" on page 2175. For a complete description of compc options, see "About the component compiler options" on page 2194.

Many command-line options, such as `show-actionscript-warnings` and `accessible`, have `true` and `false` values. You specify these values by using the following syntax:

```
mxmlc -accessible=true -show-actionscript-warnings=true
```

Some options, such as `source-path`, take a list of one or more options. You can see which options take a list by examining the help output. Square brackets (`[ ]`) that surround options indicate that the option can take a list of one or more parameters.

You can separate each entry in a list with a space or a comma. The syntax is as follows:

```
-var val1 val2
```

or

```
-var=val1, val2
```

If you do not use commas to separate entries, you terminate a list by using a double hyphen, as the following example shows:

```
-var val1 val2 -- -next_option
```

If you use commas to separate entries, you terminate a list by not using a comma after the last entry, as the following example shows:

```
-var=val1, val2 -next_option
```

You can append values to an option using the += operator. This adds the new entry to the end of the list of existing entries rather than replacing the existing entries. The following example adds the c:/myfiles directory to the `library-path` option:

```
mxmlc -library-path+=c:/myfiles
```

Using abbreviated option names

In some cases, the command-line help shows an option with dot-notation syntax; for example, `source-path` is shown as `compiler.source-path`. This notation indicates how you would set this option in a configuration file. On the command line, you can specify the option with only the final node, `source-path`, as long as that node is unique, as the following example shows:

```
mxmlc -source-path . c:/myclasses/ -- foo.mxml
```

For more information about using configuration files to store command-line options, see “About configuration files” on page 2171.

Some compiler options have aliases. *Aliases* provide shortened variations of the option name to make command lines more readable and less verbose. For example, the alias for the `output` option is `o`. You can view a list of options by their aliases by using the following command:

```
mxmlc -help list aliases
```

or

```
mxmlc -help list advanced aliases
```

You can also see the aliases in the verbose help output by using the following command:

```
mxmlc -help list details
```

Invoking the command-line compilers with Java

Flex provides a simple interface to the command-line compilers. For UNIX users, there is a shell script. For Windows users, there is an executable file. These files are located in the bin directory. You can also invoke the compilers using Java. This lets you integrate the compilers into Java-based projects (such as Ant) or other utilities.

The shell scripts and executable files for the command-line compilers wrap calls to the mxmlc.jar and compc.jar JAR files. To invoke the compilers from Java, you call the JAR files directly. For Flex SDK, the JAR files are located in the *flex_install_dir*/lib directory. For Flash Builder, they are located in the *flash_builder_install_dir*/sdks/*sdk_version*/lib.

To invoke a command in a JAR file, use the `java` command from the command line and specify the JAR file you want to execute with the `jar` option. You must also specify the value of the `+flexlib` option. This advanced option lets you set the root directory that is used by the compiler to locate the flex-config.xml file, among other files. You typically point it to your frameworks directory. From there, the compiler can detect the location of other configuration files.

The following example compiles MyApp.mxml into a SWF file using the JAR file to invoke the mxmlc compiler:

```
java -jar ../lib/mxmlc.jar +flexlib c:/flex_4_sdk/frameworks c:/flex/MyApp.mxml
```

You pass all other options as you would when you open the command-line compilers. The following example sets the locale and source path when compiling MyApp:

```
java -jar ../lib/mxmlc.jar +flexlib c:/flex_4_sdk/frameworks -locale en_US -source-path
locale/{locale} c:/flex/MyApp.mxml
```

About configuration files

Configuration files can be used by the command-line utilities and Flash Builder compilers.

Flex includes a default configuration file named flex-config.xml. This configuration file contains most of the default compiler settings for the application and component compilers. You can customize this file or create your own custom configuration file.

Flex SDK includes the flex-config.xml file in the *flex_install_dir*/frameworks directory.

The Flash Builder compilers do not use a flex-config.xml file by default. The default settings are stored internally. You can, however, create a custom configuration file and pass it to the Flash Builder compilers by using the `load-config` option. Flash Builder includes a copy of the flex-config.xml file that you can use as a template for your custom configuration file. This file located in the *flash_builder_install_dir*/sdks/*sdk_version*/frameworks directory.

You can generate a configuration file with the current settings by using the `dump-config` option, as the following example shows:

```
mxmlc -dump-config myapp-config.xml
```

Locating configuration files

You can specify the location of a configuration file by using the `load-config` option. The target configuration file can be the default flex-config.xml file, or it can be a custom configuration file. The following example loads a custom configuration file:

```
compc -load-config=myconfig.xml
```

If you specify the filename with the `+=` operator, your loaded configuration file is used *in addition to* and not instead of the flex-config.xml file:

```
compc -load-config+=myconfig.xml
```

With the mxmlc compiler, you can also use a local configuration file. A *local configuration file* does not require you to point to it on the command line. Rather, Flex examines the same directory as the target MXML file for a configuration file with the same name (one that matches the *filename*-config.xml filename). If it finds a file, it uses it in conjunction with the flex-config.xml file. You can also specify a configuration file by using the `load-config` option with the `+=` operator.

For example, if your application’s top-level file is called MyApp.mxml, the compiler first checks for a MyApp-config.xml file for configuration settings. With this feature, you can easily compile multiple applications using different configuration options without changing your command-line options or your flex-config.xml file.

Options in the local configuration file take precedence over options set in the flex-config.xml file. Options set in a configuration file that the `load-config` option specify take precedence over the local configuration file. Command-line settings take precedence over all configuration file settings. For more information on the precedence of compiler options, see “About option precedence” on page 2174.

Configuration file syntax

You store values in a configuration file in XML blocks, which follow a specific syntax. In general, the tags you use match the command-line options.

About the root tag

The root tag of the default configuration file, flex-config.xml, is `<flex-config>`. If you write a custom configuration file, it must also have this root tag. Compiler configuration files must also have an XML declaration tag, as the following example shows:

```
<?xml version="1.0"?>
<flex-config>
```

You must close the `<flex-config>` tag as you would any other XML tag. All compiler configuration files must be closed with the following tag:

```
</flex-config>
```

In general, the second tag in a configuration file is the `<compiler>` tag. This tag wraps most compiler options. However, not all compiler options are set in the `<compiler>` block of the configuration file.

Tags that you must wrap in the compiler block are prefixed by `compiler` in the help output (for example, `compiler.services`). If the option uses no dot-notation in the help output (for example, `include-file`), it is a tag at the root level of the configuration file, and the entry appears as follows:

```
<compiler>
...
</compiler>
<include-file>
    <name>logo.gif</name>
    <path>c:/images/logo/logo1.gif</path>
</include-file>
```

In some cases, options have multiple parent tags, as with the fonts options, such as `compiler.fonts.managers` and `compiler.fonts.languages.language`. Other options that require parent tags when added to a configuration file include the `frames.frame` option and the metadata options. The following sections describe methods for determining the syntax.

Getting the configuration file tags

Use the `help list` option of the command-line compilers to get the configuration file syntax of the compiler options; for example:

```
mxmlc -help list advanced
```

The following is the entry for the `source-path` option:

```
-compiler.source-path [path-element][...]
```

This indicates that in the configuration file, you can have one or more `<path-element>` child tags of the `<source-path>` tag, and that `<source-path>` is a child of the `<compiler>` tag. The following example shows how this should appear in the configuration file:

```
<compiler>
    <source-path>
        <path-element>.</path-element>
        <path-element>c:/myclasses/</path-element>
    </source-path>
</compiler>
```

Understanding leaf nodes

The help output uses dot-notation to separate child tags from parent tags, with the right-most entry being known as the *leaf node*. For example, `-tag1.tag2` indicates that `<tag2>` should be a child tag of `<tag1>`.

Angle brackets (`< >`) or square brackets (`[ ]`) that surround an option indicate that the option is a leaf node.

Square brackets indicate that there can be a list of one or more parameters for that option.

If the leaf node of a tag in the angle bracket is unique, you do not have to specify the parent tags in the configuration file. For example, the help usage shows the following:

```
compiler.fonts.managers [manager-class][...]
```

You can specify the value of this option in the configuration file, as the following example shows:

```
<compiler>
    <fonts>
        <managers>
            <manager-class>flash.fonts.JREFontManager</manager-class>
        </managers>
    </fonts>
</compiler>
```

However, the `<manager-class>` leaf node is unique, so you can set the value without specifying the `<fonts>` and `<managers>` parent tags, as the following example shows:

```
<compiler>
    <manager-class>flash.fonts.JREFontManager</manager-class>
</compiler>
```

If the help output shows multiple options listed in angle brackets, you set the values of these options at the same level inside the configuration file and do not make them child tags of each other. For example, the usage for default-size (`default-size <width> <height>`) indicates that the default size of the application is set in a configuration file, as the following example shows:

```
<default-size>
    <height>height_value</height>
    <width>width_value</width>
</default-size>
```

Using tokens

You can pass custom token values to the compiler using the following syntax:

```
+token_name=value
```

In the configuration file, you reference that value using the following syntax:

```
${token_name}
```

You can use the `@Context` token in your configuration files to represent the context root of the application. You can also use the `${flexlib}` token to represent the frameworks directory. This is useful if you set up your own configuration and are not using the default `library-path` settings.

The default value of the `${flexlib}` token is *`application_home`*`\frameworks`.

Appending values

In a configuration file, you can specify the `append` attribute of any tag that takes a list of arguments. Set this attribute to `true` to indicate that the values should be appended to the option rather than replace it. The default value is `false`.

Setting the `append` attribute to `true` lets you compound the values of options with multiple configuration files. The following example appends two entries to the `library-path` option:

```
<library-path append="true">
    <path-element>/mylibs</path-element>
    <path-element>/myotherlibs</path-element>
</library-path>
```

About option precedence

You can set the same options in multiple places and the Flex compilers use the value from the source that has the highest precedence.

If you do not specify an option on the command line, the compilers check for a `load-config` option and get the value from that file.

When using the mxmlc compiler, Flex checks to see if there is an *app_name*-config.xml file in the same directory as the target MXML file. This is known as the local configuration file and is described in "Locating configuration files" on page 2171. The syntax and structure of local configuration files are the same as with the flex-config.xml file.

If no `load-config` option is specified, the compilers check for the flex-config.xml file. The compilers look for the flex-config.xml file in the /frameworks directory. The following location is the default:

```
{flex_root}/frameworks
```

Most options have a default value that the compilers use if the option is not set in any of the other ways.

The following table shows the possible sources of options for each compiler. The table also shows the order of precedence for each option. The options set using the method described in a lower row take precedence over the options set using the methods described in a higher row.

Compiler options	Flash Builder	mxmlc	compc
Default settings	Yes	No	No
flex-config.xml	No	Yes	Yes
Local configuration file	No	Yes	No
Configuration file specified by load-config option	Yes	Yes	Yes
Command-line option	No	Yes	Yes
Options panel	Yes	No	No

You can mix and match the source of the compiler options. For example, you can specify a custom configuration file with the `load-config` option, and also set additional options on the command line.

You can also use multiple configuration files. You can chain them by using the `+=` operator with the `load-config` option. If you specify a configuration file with this option, the compilers also look for the flex-config.xml and local (*appname*-config.xml) configuration files.

Using mxmlc, the application compiler

You use the application compiler to compile SWF files from your ActionScript and MXML source files.

The application compiler's options let you define settings such as the library path and whether to include debug information in the resulting SWF file. Also, you can set application-specific settings such as the frame rate at which the SWF file should play and its height and width.

To invoke the application compiler with Flex SDK, you use the mxmlc command-line utility. In Flash Builder, you use the application compiler by building a new Flex Project. Some of the Flex SDK command-line options have equivalents in the Flash Builder environment. For example, you can use the tabs in the Flex Build Path dialog box to add classes and libraries to your project.

The following set of examples use the application compiler.

About the application compiler options

The following table describes the application compiler options. You invoke the application compiler with the mxmlc command-line utility or when building a project in Flash Builder.

Option	Description
`accessible=true|false`	Enables accessibility features when compiling the application or SWC file. The default value is `false`. For more information on using the Flex accessibility features, see "Accessible applications" on page 2122.
`actionscript-file-encoding` *`string`*	Sets the file encoding for ActionScript files. For more information, see "Setting the file encoding" on page 2190.
`advanced`	Lists advanced help options when used with the help option, as the following example shows: `mxmlc -help advanced` This is an advanced option.
`allow-source-path-overlap=true|false`	Checks if a `source-path` entry is a subdirectory of another `source-path` entry. It helps make the package names of MXML components unambiguous. This is an advanced option.
`as3=true|false`	Use the ActionScript 3.0 class-based object model for greater performance and better error reporting. In the class-based object model, most built-in functions are implemented as fixed methods of classes. The default value is `true`. If you set this value to `false`, you must set the `es` option to `true`. This is an advanced option.
`benchmark=true|false`	Prints detailed compile times to the standard output. The default value is `true`.
`compress=true|false`	Enables or disables SWF file compression. The default value is `true` when debugging is disabled (so that release SWF files are compressed) and `false` when debugging is enabled (so that debug SWF files are not compressed).
`context-root` *`context-path`*	Sets the value of the `{context.root}` token, which is often used in channel definitions in the flex-services.xml file and other settings in the flex-config.xml file. The default value is null.
`contributor` *`name`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.
`creator` *`name`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.
`date` *`text`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.

Option	Description
`debug=true|false`	Generates a debug SWF file. This file includes line numbers and filenames of all the source files. When a run-time error occurs, the stacktrace shows these line numbers and filenames. This information is used by the command-line debugger and the Flex Builder debugger. Enabling the `debug` option generates larger SWF files and also disables compilation optimization. For the mxmlc compiler, the default value is `false`. For the compc compiler, the default value is `true`. For Flash Builder, the default value is `true`. If you export an application by using the Export Release Build feature, the Flash Builder compiler excludes the debugging information, which is the equivalent of setting the value of this option to `false`. When generating SWC files, if `debug` is set to `true`, then the library.swf file inside the SWC file contains debug information. If you are generating a SWC file for distribution, set this value to `false`. For information about the command-line debugger, see "Command-line debugger" on page 2209. If you set this option to `true`, Flex also sets the `verbose-stacktraces` option to `true`
`debug-password` *`string`*	Lets you engage in remote debugging sessions with the Flash IDE. This is an advanced option.
`default-frame-rate` *`int`*	Sets the application's frame rate. The default value is 24. This is an advanced option.
`default-script-limits` *`max-recursion-depth max-execution-time`*	Defines the application's script execution limits. The `max-recursion-depth` value specifies the maximum depth of Adobe Flash Player call stack before Flash Player stops. This is essentially the stack overflow limit. The default value is 1000. The `max-execution-time` value specifies the maximum duration, in seconds, that an ActionScript event handler can execute before Flash Player assumes that it is hung, and aborts it. The default value is 60 seconds. You cannot set this value above 60 seconds. You can override these settings in the application. This is an advanced option.
`default-size` *`width height`*	Defines the default application size, in pixels. This is an advanced option.
`defaults-css-files` *`filename`* `[, ...]`	Inserts CSS files into the output the same way that a per-SWC defaults.css file works, but without having to re-archive the SWC file to test each change. CSS files included in the output with this option have a higher precedence than default CSS files in existing SWCs. For example, a CSS file included with this option overrides definitions in framework.swc's defaults.css file, but it has the same overall precedence as other included CSS files inside the SWC file. This option does not actually insert the CSS file into the SWC file; it simulates it. When you finish developing the CSS file, you should rebuild the SWC file with the new integrated CSS file. This option takes one or more files. The precedence for multiple CSS files included with this option is from first to last. This is an advanced option.

Option	Description
`defaults-css-url` *`string`*	Defines the location of the default style sheet. Setting this option overrides the implicit use of the defaults.css style sheet in the framework.swc file. For more information on the defaults.css file, see "Styles and themes" on page 1492. This is an advanced option.

`define=`*`NAMESPACE::variable,value`*	Defines a global constant. The value is evaluated at compile time and exists as a constant within the application. A common use of inline constants is to set values that are used to include or exclude blocks of code, such as debugging or instrumentation code. This is known as conditional compilation.
	The following example defines the constant `debugging` in the `CONFIG` namespace:
	`-define=CONFIG::debugging,true`
	In ActionScript, you can use this value to conditionalize statements; for example:
	`CONFIG::debugging {// Execute debugging code here.}`
	To set multiple conditionals on the command-line, use the `define` option more than once.
	For more information, see "Using conditional compilation" on page 2191.

`description` *`text`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.
`dump-config` *`filename`*	Outputs the compiler options in the flex-config.xml file to the target path; for example: `mxmlc -dump-config myapp-config.xml` This is an advanced option.
`es=true|false`	Instructs the compiler to use the ECMAScript edition 3 prototype-based object model to allow dynamic overriding of prototype properties. In the prototype-based object model, built-in functions are implemented as dynamic properties of prototype objects. The default value is `false`. Using the ECMAScript edition 3 prototype-based object model lets you use untyped properties and functions in your application code. As a result, if you set the value of the `es` compiler option to `true`, you must set the `strict` compiler option to `false`. Otherwise, the compiler will throw errors. If you set this option to `true`, you must also set the value of the `as3` compiler option to `false`. This is an advanced option.
`externs` *`class_name [...]`*	Sets a list of classes to exclude from linking when compiling a SWF file. This option provides compile-time link checking for external references that are dynamically linked. For more information about dynamic linking, see "About linking" on page 254. This is an advanced option.

Option	**Description**
`external-library-path` *`path-element [...]`*	Specifies a list of SWC files or directories to exclude from linking when compiling a SWF file. This option provides compile-time link checking for external components that are dynamically linked. By default, the libs/player/playerglobal.swc file is linked as an external library. This library is built into Flash Player. For more information about dynamic linking, see “About linking” on page 254. You can use the += operator to append the new SWC file to the list of external libraries.
`fonts.advanced-anti-aliasing=true|false`	Sets the default value that determines whether embedded fonts use advanced anti-aliasing information when rendering the font. Setting the value of the `advanced-anti-aliasing` property in a style sheet overrides this value. The default value is `false`. For more information about using advanced anti-aliasing, see “Fonts” on page 1568.
`fonts.languages.language-range` *`lang range`*	Specifies the range of Unicode settings for that language. For more information, see “Fonts” on page 1568. This is an advanced option.
`fonts.local-fonts-snapshot` *`path_to_file`*	Sets the location of the local font snapshot file. The file contains system font data. This is an advanced option.
`fonts.managers` *`manager-class [...]`*	Defines the font manager. The default is flash.fonts.JREFontManager. You can also use the flash.fonts.BatikFontManager. For more information, see “Fonts” on page 1568. This is an advanced option.
`fonts.max-cached-fonts` *`string`*	Sets the maximum number of fonts to keep in the server cache.
`fonts.max-glyphs-per-face` *`string`*	Sets the maximum number of character glyph-outlines to keep in the server cache for each font face. This is an advanced option.
`frames.frame=`*`label,class_name`*`[,...]`	Specifies a SWF file frame label with a sequence of class names that are linked onto the frame; for example: `-frame=MyLabel,Class1,Class2` This option lets you add asset factories that stream in after the application that then publish their interfaces with the ModuleManager class. The advantage to doing this is that the application starts faster than it would have if the assets had been included in the code, but does not require moving the assets to an external SWF file. This is an advanced option.
`generate-frame-loader=true|false`	Toggles the generation of an IFlexBootstrap-derived loader class. This is an advanced option.

Option	**Description**
`headless-server=true|false`	Enables the headless implementation of the Flex compiler. This sets the following: `System.setProperty("java.awt.headless", "true")` The headless setting (java.awt.headless=true) is required to use fonts and SVG on UNIX systems without X Windows. This is an advanced option.
`help [-list [advanced]]`	Prints usage information to the standard output. For example: `help -list advanced` For more information, see “Command-line syntax” on page 2169.

`include-inheritance-dependencies-only=false|true`	Include only classes that are inheritance dependencies of classes that are included with the `include-classes` compiler option.
	The default value is `false`.
	This is an advanced option. You might use this compiler option if you are creating a custom RSL and want to externalize as many classes as possible. For example:
	`compc -include-classes mx.collections.ListCollectionView`
	`-include-inheritance-dependencies-only=true`
	`-source-path . -output lcv2 -directory`

`include-libraries` *`library`* `[...]`	Links all classes inside a SWC file to the resulting application SWF file, regardless of whether or not they are used. Contrast this option with the `library-path` option that includes only those classes that are referenced at compile time. To link one or more classes whether or not they are used and not an entire SWC file, use the `includes` option. This option is commonly used to specify resource bundles.
`include-resource-bundles` *`bundle [...]`*	Specifies the resource bundles to link into a resource module. All resource bundles specified with this option must be in the compiler’s source path. You specify this using the `source-path` compiler option. For more information on using resource bundles, see “Resource Bundles” on page 2091.
`includes` *`class`* `[...]`	Links one or more classes to the resulting application SWF file, whether or not those classes are required at compile time. To link an entire SWC file rather than individual classes, use the `include-libraries` option.
`incremental=true|false`	Enables incremental compilation. For more information, see “About incremental compilation” on page 2191. This option is `true` by default for the Flash Builder application compiler. For the command-line compiler, the default is `false`.
`isolate-styles=true|false`	Enables per-module styling. The default value is `true`. This is an advanced option. You typically only use this option if you want to set styles in a module that is loaded into your main application. For more information about modules, see “Modular applications” on page 138.

Option	**Description**
`keep-as3-metadata=class_name [...]`	Specifies custom metadata that you want to keep. By default, the compiler keeps the following metadata: • Bindable • Managed • ChangeEvent • NonCommittingChangeEvent • Transient If you want to preserve the default metadata, you should use the += operator to append your custom metadata, rather than the = operator which replaces the default metadata. This is an advanced option. For more information, see "About metadata tags" on page 2376.
`keep-all-type-selectors=true|false`	Instructs the compiler to keep a style sheet's type selector in a SWF file, even if that type (the class) is not used in the application. This is useful when you have a modular application that loads other applications. For example, the loading SWF file might define a type selector for a type used in the loaded (or, target) SWF file. If you set this option to `true` when compiling the loading SWF file, then the target SWF file will have access to that type selector when it is loaded. If you set this option to `false`, the compiler will not include that type selector in the loading SWF file at compile time. As a result, the styles will not be available to the target SWF file. This is an advanced option.
`keep-generated-actionscript=true|false`	Determines whether to keep the generated ActionScript class files. The generated class files include stubs and classes that are generated by the compiler and used to build the SWF file. When using the application compiler, the default location of the files is the /generated subdirectory, which is directly below the target MXML file. If the /generated directory does not exist, the compiler creates one. When using the compc component compiler, the default location of the /generated directory is relative to the output of the SWC file. When using Flash Builder, the default location of the generated files is the /bin/generated directory. The default names of the primary generated class files are filename-generated.as and filename-interface.as. The default value is `false`. This is an advanced option.
`language code`	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.

Option	Description
`library-path` *`path-element [...]`*	Links SWC files to the resulting application SWF file. The compiler only links in those classes for the SWC file that are required. You can specify a directory or individual SWC files. The default value of the `library-path` option includes all SWC files in the libs and libs/player directories, plus the current locale directory. These are required. To point to individual classes or packages rather than entire SWC files, use the `source-path` option. If you set the value of the `library-path` as an option of the command-line compiler, you must also explicitly add the framework.swc and locale SWC files. Your new entry is not appended to the `library-path` but replaces it, unless you use the += operator. On the command line, you use the += operator to append the new argument to the list of existing SWC files. In a configuration file, you can set the `append` attribute of the `library-path` tag to `true` to indicate that the values should be appended to the library path rather than replace existing default entries.
`license` *`product_name  license_key`*	Defines the license key to use when compiling.
`link-report` *`filename`*	Prints linking information to the specified output file. This file is an XML file that contains `<def>`, `<pre>`, and `<ext>` symbols showing linker dependencies in the final SWF file. The output of this command can be used as input to the `load-externs` option. For more information on the report, see "Examining linker dependencies" on page 2308. This is an advanced option.
`load-config` *`filename`*	Specifies the location of the configuration file that defines compiler options. If you specify a configuration file, you can override individual options by setting them on the command line. All relative paths in the configuration file are relative to the location of the configuration file itself. Use the `+=` operator to chain this configuration file to other configuration files. For more information on using configuration files to provide options to the command-line compilers, see "About configuration files" on page 2171.
`load-externs` *`filename [...]`*	Specifies the location of an XML file that contains `<def>`, `<pre>`, and `<ext>` symbols to omit from linking when compiling a SWF file. The XML file uses the same syntax as the one produced by the `link-report` option. For more information on the report, see "Examining linker dependencies" on page 2308. This option provides compile-time link checking for external components that are dynamically linked. For more information about dynamic linking, see "About linking" on page 254. This is an advanced option.

Option	Description
`locale locale[,...]`	Specifies one or more locales to be compiled into the SWF file. If you do not specify a locale, then the compiler uses the default locale from the flex-config.xml file. The default value is en_US. You can append additional locales to the default locale by using the += operator. If you remove the default locale from the flex-config.xml file, and do not specify one on the command line, then the compiler will use the machine's locale. For more information, see "Localization" on page 2055.
`localized-description text lang`	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.
`localized-title text lang`	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.
`mxml.compatibility-version=version`	Specifies the version of the Flex compiler that the output should be compatible with. This option affects some behavior such as the layout rules, padding and gaps, skins, and other style settings. In addition, it affects the rules for parsing properties files. The following example instructs the application to compile with the Flex 3 rules for these behaviors: `-compatibility-version=3.0.0` Possible values for this compiler option are defined as constants in the FlexVersion class. If you set this value to 3.0.0, you must also use a theme in your application that is compatible with version 3 of the SDK. For more information on using this option, see "Backward compatibility" on page 2239.

`mxml.minimum-supported-`
`version=`***`version_number`***
	Specifies the minimum version of the SDK that the application uses. This is typically used when generating SWC files.
	This is an advanced option.

`mxml.qualified-type-selectors=true|false`	Determines whether you want the compiler to ensure that type selectors have a qualified namespace in the CSS files. This is an advanced option. The default value is `true`.
`namespaces.namespace uri manifest`	Specifies a namespace for the MXML file. You must include a URI and the location of the manifest file that defines the contents of this namespace. This path is relative to the MXML file. For more information about manifest files, see "About manifest files" on page 2206.
`optimize=true|false`	Enables the ActionScript optimizer. This optimizer reduces file size and increases performance by optimizing the SWF file's bytecode. The default value is `true`.
`omit-trace-statements=false|true`	Enables `trace()` statements from being written to the flashlog.txt file. The default value is `false`, which means that by default, `trace()` statements are written to the flashlog.txt file.

Option	Description
`output` *`filename`*	Specifies the output path and filename for the resulting file. If you omit this option, the compiler saves the SWF file to the directory where the target file is located. The default SWF filename matches the target filename, but with a SWF file extension. If you use a relative path to define the filename, it is always relative to the current working directory, not the target MXML application root. The compiler creates extra directories based on the specified filename if those directories are not present. When using this option with the component compiler, the output is a SWC file rather than a SWF file, unless you set the `directory` option to `true`. In that case, the output is a directory with the contents of the SWC file. The name of the directory is that value of the `ouput` option.
`preloader` ***`class_name`***	Specify a download progress bar for your application. The value must be the name of a class that implements the IPreloaderDisplay interface. The default value is "mx.preloaders.SparkDownloadProgressBar" when `compatibility-version` is 4.0.0 or greater. When `compatibility-version` is less than 4.0.0, the default value is "mx.preloaders.DownloadProgressBar". For more information, see "Showing the download progress of an application" on page 408.
`publisher` *`name`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.
`raw-metadata` *`XML_string`*	Defines the metadata for the resulting SWF file. The value of this option overrides any metadata-related compiler options such as `contributor`, `creator`, `date`, and `description`. This is an advanced option.
`remove-unused-rsls=true|false`	Instructs the compiler to only include RSLs that are used by the application. The default value is `true`. For more information about RSLs, see "Runtime Shared Libraries" on page 253.
`resource-bundle-list` *`filename`*	Prints a list of resource bundles that are used by the current application to a file named with the ***`filename`*** argument. You then use this list as input that you specify with the `include-resource-bundles` option to create a resource module. For more information, see "Resource Bundles" on page 2091.
`runtime-shared-libraries` *`rsl-url [...]`*	Specifies a list of runtime shared libraries (RSLs) to use for this application. RSLs are dynamically-linked at run time. The compiler externalizes the contents of the application that you are compiling that overlap with the RSL. You specify the location of the SWF file relative to the deployment location of the application. For example, if you store a file named library.swf file in the *web_root*/libraries directory on the web server, and the application in the web root, you specify libraries/library.swf. This compiler argument is included for backwards compatibility with Flex 3 applications. For Flex 4 applications, use the `runtime-shared-library-path` option. For more information about RSLs, see "Runtime Shared Libraries" on page 253.

Option	Description
`runtime-shared-library-path=`*`path-element`*`,`*`rsl-url`*`[,`*`policy-file-url`*`,`*`failover-url`*`,...]`	Specifies the location of a runtime shared library (RSL). The compiler externalizes the contents of the application that you are compiling that overlap with the RSL. The `path-element` argument is the location of the SWC file or open directory to compile against. For example, c:\flexsdk\frameworks\libs\framework.swc. This is the equivalent of the using the `external-library-path` option when compiling against an RSL using the `runtime-shared-libraries` option. The `rsl-url` argument is the URL of the RSL that will be used to load the RSL at runtime. The compiler does not verify the existence of the SWF file at this location at compile time. It does store this string in the application, however, and uses it at run time. As a result, the SWF file must be available at run time but necessarily not at compile time. The `policy-file-url` is the location of the crossdomain.xml file that gives permission to read the RSL from the server. This might be necessary because the RSL can be on a separate server as the application. For example, http://www.mydomain.com/rsls/crossdomain.xml. The `failover-url` and second `policy-file-url` arguments specify the location of the secondary RSL and crossdomain.xml file if the first RSL cannot be loaded. This most commonly happens when the client Player version does not support cross-domain RSLs. You can add any number of failover RSLs, but must include a policy file URL for each one. Do not include spaces between the comma-separated values. The following example shows how to use this option: `mxmlc -o=../lib/app.swf -runtime-shared-library-path=../lib/mylib.swc,../bin/myrsl.swf Main.mxml` You can specify more than one library file to be used as an RSL. You do this by adding additional `runtime-shared-library-path` options. You can also use the `runtime-shared-libraries` command to use RSLs with your applications. However, the `runtime-shared-library-path` option lets you also specify the location of the policy file and failover RSL. For more information about RSLs, see "Runtime Shared Libraries" on page 253.
`runtime-shared-library-settings.application-domain=`***`path-element`***`,`***`application-domain-target`***	Controls which domain an RSL is loaded into at runtime. The `path-element` is the path of the SWC library. To specify an RSL with this option, you must also define it in the `runtime-shared-library-path` option. The `application-domain-target` is the domain that the RSL should be loaded into. Valid values for `application-domain-target` are `default`, `current`, `parent`, and `top-level`. The default value is `default`. For more information about modules, see "Modular applications" on page 138. For more information about sub-applications, see "Developing and loading sub-applications" on page 176. For more information about RSLs, see "Runtime Shared Libraries" on page 253.

Option	Description
`runtime-shared-library-settings.force-rsls=`***`path-element`***	Forces an RSL to be included, regardless of whether it is used by the application. This is useful if all the links to a particular class in an RSL are soft references, or if you anticipate a module or sub-application to need a class that is not used by the main application. The `path-element` is the path of the SWC library. You can specify more than one `path-element` by using a comma-delimited list of RSLs. To specify an RSL with this option, you must also define it in the `runtime-shared-library-path` option. For more information about RSLs, see "Runtime Shared Libraries" on page 253.
`services` *`filename`*	Specifies the location of the services-config.xml file. This file is used by LiveCycle Data Services ES.
`show-actionscript-warnings=true|false`	Shows warnings for ActionScript classes. The default value is `true`. For more information about viewing warnings and errors, see "Viewing warnings and errors" on page 2203.
`show-binding-warnings=true|false`	Shows a warning when Flash Player cannot detect changes to a bound property. The default value is `true`. For more information about viewing warnings and errors, see "Viewing warnings and errors" on page 2203.
`show-invalid-css-property-warnings=true|false`	Shows a warning when a style property is set in CSS on a component that does not support that property. The warning can be a result of the theme not supporting the style property, the component not declaring it, or the component excluding it. The default value is `true`. For more information about viewing warnings and errors, see "Viewing warnings and errors" on page 2203.
`show-shadowed-device-font-warnings=true|false`	Shows warnings when you try to embed a font with a family name that is the same as the operating system font name. The compiler normally warns you that you are shadowing a system font. Set this option to `false` to disable the warnings. The default value is `true`. For more information about viewing warnings and errors, see "Viewing warnings and errors" on page 2203.
`show-unused-type-selector-warnings=true|false`	Shows warnings when a type selector in a style sheet or `<fx:Style>` block is not used by any components in the application. The default value is `true`. This warning does not detect whether a condition is met for descendant selectors. For more information about viewing warnings and errors, see "Viewing warnings and errors" on page 2203.

Option	Description
`size-report=`***`filename`***	Creates a report that summarizes the size of each type of data within an application SWF file. Types of data include media files, fonts, shapes, and ActionScript. For example: `mxmlc -size-report=myreport.xml MyApp.mxml` The file format of the output is XML. While the format has some similarities to the output of the `link-report` option, the size report cannot be used as input to the `load-externs` option. This option is useful if you are optimizing your application and want to see how big areas of your application are. For more information about the size report, see "Using the size report" on page 2305.
`source-path` *`path-element [...]`*	Adds directories or files to the source path. The Flex compiler searches directories in the source path for MXML, AS, or CSS source files that are used in your applications and includes those that are required at compile time. You can use wildcards to include all files and subdirectories of a directory. To link an entire library SWC file and not individual classes or directories, use the `library-path` option. The source path is also used as the search path for the component compiler's `include-classes` and `include-resource-bundles` options. You can also use the += operator to append the new argument to the list of existing source path entries. This option has the following default behavior: • If `source-path` is empty, the target file's directory will be added to source-path. • If `source-path` is not empty and if the target file's directory is a subdirectory of one of the directories in `source-path`, `source-path` remains unchanged. • If `source-path` is not empty and if the target file's directory is not a subdirectory of any one of the directories in `source-path`, the target file's directory is prepended to `source-path`.
`static-link-runtime-shared-libraries=true|false`	Determines whether to compile against libraries statically or use RSLs. Set this option to `true` to ignore the RSLs specified by the `runtime-shared-library-path` option. Set this option to `false` to use the RSLs. The default value is `true`. This option is useful so that you can quickly switch between a statically and dynamically linked application without having to change the `runtime-shared-library-path` option, which can be verbose, or edit the configuration files. For more information about RSLs, see "Runtime Shared Libraries" on page 253.
`strict=true|false`	Prints undefined property and function calls; also performs compile-time type checking on assignments and options supplied to method calls.\ The default value is `true`. For more information about viewing warnings and errors, see "Viewing warnings and errors" on page 2203.

Option	**Description**
`swf-version=`***`int`***	Specifies the SWF file format version of the output SWF file. Features requiring a later version of the SWF file format are not compiled into the application. This is different from the Player version in that it refers to the SWF specification versioning scheme. For Flex 4.6, the default value of `swf-version` is 14 (and the default Player version is 11.1). This is an advanced option. For more information, see "Targeting Flash Player versions" on page 2240.
`target-player=`*`major_version[.minor_version.revision]`*	Specifies the version of Flash Player that you want to target with the application. Features requiring a later version of Flash Player are not compiled into the application. The ***`player_version`*** parameter has the following format: *`major_version.minor_version.revision`* The ***`major_version`*** is required while ***`minor_version`*** and ***`revision`*** are optional. For Flex 4.0 and 4.1, the minimum value is 10.0.0. If you do not specify the ***`minor_version`*** or ***`revision`***, then the compiler uses zeros. For Flex 4.5, the default value is 10.2.0. For Flex 4.6, the default value is 11.1. If you do not explicitly set the value of this option, the compiler uses the default from the flex-config.xml file. The value in flex-config.xml is the version of Flash Player that shipped with the SDK. This option is useful if your application's audience has a specific Player and cannot upgrade. You can use this option to "downgrade" your application for that audience. For more information, see "Targeting Flash Player versions" on page 2240
`theme` *`filename [...]`*	Specifies a list of theme files to use with this application. Theme files can be SWC files with CSS files inside them or CSS files. For information on compiling a SWC theme file, see "Styles and themes" on page 1492.
`title` *`text`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 2189.
`tools-locale=`***`locale`***	Specifies the locale to use when reporting compiler errors and warnings. Valid values are the language code (such as "ja" or "en") or the language code plus country code (such as "ja_JP" or "en_US"), depending on your system's configuration.
`use-direct-blit=false|true`	Specifies whether hardware acceleration is used to copy graphics to the screen (if such acceleration is available). This option only applies to applications running in the standalone Flash Player. For applications running in a browser, setting `use-direct-blit` to `true` is equivalent to setting `wmode` to `"direct"` in the HTML wrapper. For AIR applications, use the `renderMode` application descriptor tag. The default value is `false`. This is an advanced option.

Option	Description
`use-gpu=false|true`	Specifies whether GPU (Graphics Processing Unit) acceleration is used when drawing graphics (if such acceleration is available). This option only applies to applications running in the standalone Flash Player. For applications running in a browser, setting `use-gpu` to `true` is equivalent to setting `wmode` to `"gpu"` in the HTML wrapper. For AIR applications, use the `renderMode` application descriptor tag. The default value is `false`. This is an advanced option.
`use-network=true|false`	Specifies that the current application uses network services. The default value is `true`. When the `use-network` property is set to `false`, the application can access the local filesystem (for example, use the `XML.load()` method with file: URLs) but not network services. In most circumstances, the value of this property should be `true`. For more information about the `use-network` property, see "Security" on page 117.
`use-resource-bundle-metadata=true|false`	Enables resource bundles. Set to `true` to instruct the compiler to process the contents of the `[ResourceBundle]` metadata tag. The default value is `true`. For more information, see "Resource Bundles" on page 2091. This is an advanced option.
`verbose-stacktraces=true|false`	Generates source code that includes line numbers. When a run-time error occurs, the stacktrace shows these line numbers. Enabling this option generates larger SWF files. Enabling this option does not generate a debug SWF file. To do that, you must set the `debug` option to `true`. The default value is `false`.
`verify-digests=true|false`	Instructs the application to check the digest of the RSL SWF file against the digest that was compiled into the application at compile time. This is a security measure that lets you load RSLs from remote domains or different sub-domains. It also lets you enforce versioning of your RSLs by forcing an application's digest to match the RSL's digest. If the digests are out of sync, you must recompile your application or load a different RSL SWF file. For more information about RSLs, see "Runtime Shared Libraries" on page 253.
`version`	Returns the version number of the MXML compiler. If you are using a trial or Beta version of Flex, the version option also returns the number of days remaining in the trial period and the expiration date.
`warn-warning_type=true|false`	Enables specified warnings. For more information, see "Viewing warnings and errors" on page 2203.
`warnings=true|false`	Enables all warnings. Set to `false` to disable all warnings. This option overrides the `warn-`***`warning_type`*** options. The default value is `true`.

The following sections provide examples of using the mxmlc application compiler options on the command line. You can also use these techniques with the application compiler in Flash Builder.

Basic example of using mxmlc

The most basic example is one in which the MXML file has no external dependencies (such as components in a SWC file or ActionScript classes) and no special options. In this case, you invoke the mxmlc compiler and point it to your MXML file as the following example shows:

```
mxmlc c:/myfiles/app.mxml
```

The default option is the target file to compile into a SWF file, and it is required to have a value. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file; for example:

```
mxmlc -option arg1 arg2 arg3 -- target_file.mxml
```

Adding metadata to SWF files

The application compilers support adding metadata to SWF files. This metadata can be used by search engines and other utilities to gather information about the SWF file. This metadata represents a subset of the Dublin Core schema.

You can set the following values:

- `contributor`
- `creator`
- `date`
- `description`
- `language`
- `localized-description`
- `localized-title`
- `publisher`
- `title`

For the mxmlc command-line compiler, the default metadata settings in the flex-config.xml file are as follows:

```
<metadata>
    <title>Adobe Flex 4 Application</title>
    <description>http://www.adobe.com/products/flex</description>
    <publisher>unknown</publisher>
    <creator>unknown</creator>
    <language>EN</language>
</metadata>
```

You can also set the metadata values as command-line options. The following example sets some of the metadata values:

```
mxmlc -language+=klingon -title "checkintest!"
    -localized-description "it r0x0rs" en-us
    -localized-description "c'est magnifique!" fr-fr
    -creator "Flexy Frank" -publisher "Franks Beans" flexstore.mxml
```

In this example, the following values are compiled into the resulting SWF file:

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <rdf:Description rdf:about='' xmlns:dc='http://purl.org/dc/elements/1.1'>
        <dc:format>application/x-shockwave-flash</dc:format>
        <dc:title>checkintest!</dc:title>
        <dc:description>
            <rdf:Alt>
                <rdf:li xml:lang='fr-fr'>c'est magnifique!</rdf:li>
                <rdf:li xml:lang='x-default'>http://www.adobe.com/products/flex</rdf:li>
                <rdf:li xml:lang='en-us'>it r0x0rs</rdf:li>
            </rdf:Alt>
        </dc:description>
        <dc:publisher>Franks Beans</dc:publisher>
        <dc:creator>Flexy Frank</dc:creator>
        <dc:language>EN</dc:language>
        <dc:language>klingon</dc:language>
        <dc:date>Mar 16, 2010</dc:date>
    </rdf:Description>
</rdf:RDF>
```

For information on the SWF file format, see the Flash File Format Specification, which is available through the Player Licensing program.

Setting the file encoding

You use the `actionscript-file-encoding` option to set the file encoding so that the application compiler correctly interprets ActionScript files. This tag does not affect MXML files because they are XML files that contain an encoding specification in the `xml` tag.

You use the `actionscript-file-encoding` option when your ActionScript files do not contain a Byte Order Mark (BOM), and the files use an encoding that is different from the default encoding of your computer. If your ActionScript files contain a BOM, the compiler uses the information in the BOM to determine the file encoding.

For example, if your ActionScript files use Shift_JIS encoding, have no BOM, and your computer uses ISO-8859-1 as the default encoding, you use the `actionscript-file-encoding` option, as the following example shows:

```
actionscript-file-encoding=Shift_JIS
```

Editing application settings

The mxmlc compiler includes options to set the application's frame rate, size, and script limits. By setting them when you compile your application, you do not need to edit the HTML wrapper or the application's MXML file. You can override these settings by using properties of the `<s:Application>` tag or properties of the `<object>` and `<embed>` tags in the HTML wrapper.

The following command-line example sets default application properties:

```
mxmlc -default-size 240 240 -default-frame-rate=24 -default-script-limits 5000 10 --
c:/myfiles/flex/misc/MainApp.mxml
```

For more information about the Application class, see "Application containers" on page 393.

For more information about the HTML wrapper, see "Creating a wrapper" on page 2552.

Using SWC files

Often, you use SWC files when compiling MXML files. SWC files can provide themes, components, or other helper files. You typically specify SWC files used by the application by using the `library-path` option.

The following example compiles the RotationApplication.mxml file into the RotationApplication.swf file:

```
mxmlc -library-path+=c:/mylibraries/MyButtonSwc.swc c:/myfiles/comptest/testRotation.mxml
```

In a configuration file, this appears as follows:

```
<compiler>
    <library-path>
        <path-element>libs</path-element>
        <path-element>libs/player</path-element>
        <path-
element>libs/player/{targetPlayerMajorVersion}.{targetPlayerMinorVersion}</path-element>
        <path-element>locale/{locale}</path-element>
        <path-element>c:/mylibraries/MyButtonSwc.swc</path-element>
    </library-path>
</compiler>
```

About incremental compilation

You can use incremental compilation to decrease the time it takes to compile an application or component library with the application compilers. When incremental compilation is enabled, the compiler inspects changes to the bytecode between revisions and only recompiles the section of bytecode that has changed. These sections of bytecode are also referred to as *compilation units*.

You enable incremental compilation by setting the `incremental` option to `true`, as the following example shows:

```
mxmlc -incremental=true MyApp.mxml
```

Incremental compilation means that the compiler inspects your code, determines which parts of the application are affected by your changes, and only recompiles the newer classes and assets. The Flex compilers generate many compilation units that do not change between compilation cycles. It is possible that when you change one part of your application, the change might not have any effect on the bytecode of another.

As part of the incremental compilation process, the compiler generates a cache file that lists the compilation units of your application and information on your application's structure. This file is located in the same directory as the file that you are compiling. For example, if my application is called MyApp.mxml, the cache file is called MyApp.mxml.cache. This file helps the compiler determine which parts of your application must be recompiled. One way to force a complete recompile is to delete the cache file from the directory.

Incremental compilation can help reduce compile time on small applications, but you achieve the biggest gains on larger applications.

The default value of the `incremental` compiler option is `true` for the Flash Builder application compiler. For the mxmlc command-line compiler, the default is `false`.

Using conditional compilation

To include or exclude blocks of code for certain builds, you can use conditional compilation. The mxmlc compiler lets you pass the values of constants to the application at compile time. Commonly, you pass a Boolean that is used to include or exclude a block of code such as debugging or instrumentation code. The following example conditionalizes a block of code by using an inline constant Boolean:

```
CONFIG::debugging {
    // Execute debugging code here.
}
```

To pass constants, you use the `compiler.define` compiler option. The constant can be a Boolean, String, or Number, or an expression that can be evaluated in ActionScript at compile time. This constant is then accessible within the application source code as a global constant.

To use the `define` option, you define a configuration namespace for the constant, a variable name, and a value using the following syntax:

```
-define=namespace::variable_name,value
```

The configuration namespace can be anything you want. The following example defines the constant `debugging` with a value of `true` in the `CONFIG` namespace:

```
-define=CONFIG::debugging,true
```

You use the `+=` operator rather than the `=` operator to append definitions on the command line to the definitions set in the configuration file. Use the `=` operator to replace the definitions in the configuration file with the definitions on the command line.

To set the values of multiple constants on the command-line, use the `define` option more than once; for example:

```
mxmlc -define+=CONFIG::debugging,true -define+=CONFIG::release,false MyApp.mxml
```

To set the value of these constants in the flex-config.xml file, rather than on the command line, you write this as the following example shows:

```
<compiler>
    <define append="true">
        <name>CONFIG::debugging</name>
        <value>true</value>
    </define>
    <define append="true">
        <name>CONFIG::release</name>
        <value>false</value>
    </define>
</compiler>
```

If you set the same definition in a configuration file and on the command line, the value on the command line takes precedence.

In a Flex Ant task, you can set constants with a `define` element, as the following example shows:

```
<mxmlc ... >
    <define name="CONFIG::debugging" value="true"/>
    <define name="CONFIG::release" value="false"/>
</mxmlc>
```

Using inline constants

You can use inline constants in ActionScript. Boolean values can be used to conditionalize top-level definitions of functions, classes, and variables, in much the same way you would use an `#IFDEF` preprocessor command in C or C++. You cannot use constant Boolean values to conditionalize metadata or `import` statements.

The following example conditionalizes which class definition the compiler uses when compiling the application:

```
// compilers/MyButton.as
package  {
    import mx.controls.Button;
    CONFIG::debugging
    public class MyButton extends Button {
        public function MyButton() {
            super();
            // Set the label text to blue.
            setStyle("color", 0x0000FF);
        }
    }
    CONFIG::release
    public class MyButton extends Button {
        public function MyButton() {
            super();
            // Set the label text to red.
            setStyle("color", 0xFF0000);
        }
    }
}
```

You can also pass Strings and Numbers to the application and use them as inline constants, in the same way you might use a `#define` directive in C or C++. For example, if you pass a value named `NAMES::Company`, you replace it with a constant in your application by using an ActionScript statement like the following example shows:

```
private static const companyName:String = NAMES::Company;
```

Passing expressions

You can pass expressions that can be evaluated at compile time as the value of the constant. The following example evaluates to false:

```
-define+=CONFIG::myConst,"1 > 2"
```

The following example evaluates to 3:

```
-define+=CONFIG::myConst,"4 - 1"
```

Expressions can contain constants and other configuration values; for example:

```
-define+=CONFIG::bool2,false -define+=CONFIG::and1,"CONFIG::bool2 && false"
```

In general, you should wrap all constants with double quotes, so that the mxmlc compiler correctly parses them as a single argument.

Passing Strings

When passing Strings, you must add extra quotes to ensure that the compiler parses them correctly.

To define Strings on the command-line, you must surround them with double-quotes, *and* either escape-quote them (`"\"Adobe Systems\""` or `"\\x\\x`) or single-quote them (`"'Adobe Systems'"`).

The following example shows both methods of including Strings on the command line:

```
-define+=NAMES::Company,"'Adobe Systems'" -define+=NAMES::Ticker,"\"ADBE\""
```

To define Strings in configuration files, you must surround them with single or double quotes; for example:

```
<define>
    <name>NAMES::Company</name>
    <value>'Adobe Systems'</value>
</define>
<define>
    <name>NAMES::Ticker</name>
    <value>"ADBE"</value>
</define>
```

To pass empty Strings on the command line, use single quotes surrounded by double quotes, as the following example shows:

```
-define+=CONFIG::debugging,"''"
```

To pass empty Strings in configuration files, use double quotes (`""`) or single quotes (`' '`).

Using compc, the component compiler

You use the component compiler to generate a SWC file from component source files and other asset files such as images and style sheets.

To use the component compiler with Flex SDK, you use the compc command-line utility. In Flash Builder, you use the compc component compiler by building a new Flex Library Project. Some of the Flex SDK command-line options have equivalents in the Flash Builder environment. You use the tabs in the Flex Library Build Path dialog box to add classes, libraries, and other resources to the SWC file.

Xavi Beumala has an excellent blog entry on creating libraries with compc.

About the component compiler options

The component compiler options let you define settings such as the classes, resources, and namespaces to include in the resulting SWC file.

The component compiler can take most of the application compiler options, and the options described in this section. For a description of the application compiler options, see “About the application compiler options” on page 2175. Application compiler options that do not apply to the component compiler include the metadata options (such as `contributor`, `title`, and `date`), default application options (such as `default-frame-rate`), `locale`, `debug-password`, and `theme`.

The component compiler has compiler options that the application compilers do not have. The following table describes the component compiler options that are not used by the application compilers:

Option	Description
`compute-digest=true|false`	Writes a digest to the catalog.xml of a library. Use this when the library will be used as a cross-domain RSL or when you want to enforce the versioning of RSLs. The default value is true. For more information about RSLs, see "Runtime Shared Libraries" on page 253.
`directory=false|true`	Outputs the SWC file in an open directory format rather than a SWC file. You use this option with the `output` option to specify a destination directory, as the following example shows: `compc -directory=true -output=destination_directory` You typically use this option when you create RSLs because you must extract the library.swf file from the SWC file before deployment. For more information, see "Runtime Shared Libraries" on page 253. The default value is false.
`include-classes` *`class [...]`*	Specifies classes to include in the SWC file. You provide the class name (for example, MyClass) rather than the file name (for example, MyClass.as) to the file for this option. As a result, all classes specified with this option must be in the compiler's source path. You specify this by using the `source-path` compiler option. You can use packaged and unpackaged classes. To use components in namespaces, use the `include-namespaces` option. If the components are in packages, ensure that you use dot-notation rather than slashes to separate package levels. This is the default option for the component compiler.
`include-file` *`name path [...]`*	Adds the file to the SWC file. This option does not embed files inside the library.swf file. This is useful for adding graphics files, where you want to add non-compiled files that can be referenced in a style sheet or embedded as assets in MXML files. If you add a stylesheet that references compiled resources such as programmatic skins, use the `include-stylesheet` option. If you use the `[Embed]` syntax to add a resource to your application, you are not required to use this option to also link it into the SWC file. For more information, see "Adding nonsource classes" on page 2202.
`include-lookup-only=false|true`	If `true`, only manifest entries with `lookupOnly=true` are included in the SWC catalog. The default value is `false`. This is an advanced option.

Option	**Description**
`include-namespaces` *`uri [...]`*	Specifies namespace-style components in the SWC file. You specify a list of URIs to include in the SWC file. The uri argument must already be defined with the `namespace` option. To use components in packages, use the `include-classes` option.
`include-sources` *`path-element`*	Specifies classes or directories to add to the SWC file. When specifying classes, you specify the path to the class file (for example, MyClass.as) rather than the class name itself (for example, MyClass). This lets you add classes to the SWC file that are not in the source path. In general, though, use the `include-classes` option, which lets you add classes that are in the source path. If you specify a directory, this option includes all files with an MXML or AS extension, and ignores all other files. If you use this option to include MXML components that are in a non-default package, you must include the source folder in the source path.
`include-stylesheet` *`namepath [...]`*	Specifies stylesheets to add to the SWC file. This option compiles classes that are referenced by the stylesheet before including the stylesheet in the SWC file. You do not need to use this option for all stylesheets; only stylesheets that reference assets that need to be compiled such as programmatic skins or other class files. If your stylesheet does not reference compiled assets, you can use the `include-file` option. This option does not compile the stylesheet into a SWF file before including it in the SWC file. You compile a CSS file into a SWF file when you want to load it at run time.

On the command line, you cannot point the compc utility to a single directory and have it compile all components and source files in that directory. You must specify each source file to compile.

If you have a large set of components in a namespace to include in a SWC file, you can use a manifest file to avoid having to type an unwieldy compc command. For information on creating manifest files, see “About manifest files” on page 2206.

The following sections describe common scenarios where you could use the compc command-line compiler. You can apply the techniques described here to compiling SWC files in Flash Builder with the Flex Compiler.

Compiling stand-alone components and classes

In many cases, you have one or more components that you use in your applications, but you do not have them in a package structure. You want to be able to use them in the generic namespace (“*”) inside your applications. In these cases, you use the `include-classes` option to add the components to your SWC file.

The following command-line example compiles two MXML components, Rotation.as and RotationInstance.as, into a single SWC file:

```
compc -source-path .
    -output c:/jrun4/servers/flex/WEB-INF/flex/user_classes/RotationClasses.swc
    -include-classes rotationClasses.Rotation rotationClasses.RotationInstance
```

The rotationClasses directory is a subdirectory of the current directory, which is in the source path. The SWC file is output to the user_classes directory, so the new components require no additional configuration to be used in a server environment.

You use the `include-classes` option to add components to the SWC file. You use just the class name of the component and not the full filename (for example, MyComponent rather than MyComponent.as). Use dot-notation to specify the location of the component in the package structure.

You also set the `source-path` to the current directory or a directory from which the component directory can be determined.

You can also add the framework.swc and framework_rb.swc files to the `library-path` option. This addition is not always required if the compiler can determine the location of these SWC files on its own. However, if you move the compiler utility out of the default location relative to the frameworks files, you must add it to the library path.

The previous command-line example appears in a configuration file as follows:

```
<compiler>
    <source-path>
        <path-element>.</path-element>
    </source-path>
    <output>
        c:/jrun4/servers/flex/WEB-INF/flex/user_classes/RotationClasses.swc
    </output>
</compiler>
<include-classes>
    <class>rotationClasses.Rotation</class>
    <class>rotationClasses.RotationInstance</class>
</include-classes>
```

To use components that are not in a package in an application, you must declare a namespace that includes the directory structure of the components. The following example declares a namespace for the components compiled in the previous example:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmln:local="rotationclasses.*">
    ...
    <local:Rotation id="Rotate75" angleFrom="0" angleTo="75" duration="100"/>
    ...
</s:Application>
```

To use the generic namespace of "*" rather than a namespace that includes a component's directory structure, you can include the directory in the `source-path` as the following command-line example shows:

```
compc -source-path . c:/flexdeploy/comps/rotationClasses
    -output c:/jrun4/servers/flex/WEB-INF/flex/user_classes/RotationComps.swc
    -include-classes Rotation RotationInstance
```

Then, you can specify the namespace in your application as:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmln:local="*">
```

You are not required to use the directory name in the `include-classes` option if you add the directory to the source path.

These options appear in a configuration file, as the following example shows:

```
<compiler>
    <source-path>
        <path-element>.</path-element>
        <path-element>c:/flexdeploy/comps/rotationClasses</path-element>
    </source-path>
    <output>c:/jrun4/servers/flex/WEB-INF/flex/user_classes/RotationComps.swc</output>
</compiler>
<include-classes>
    <class>Rotation</class>
    <class>RotationInstance</class>
<include-classes>
```

This example assumes that the components are not in a named package. For information about compiling packaged components, see “Compiling components in packages” on page 2198.

Compiling components in packages

Some components are created inside packages or directory structures so that they can be logically grouped and separated from application code. As a result, packaged components can have a namespace declaration that includes the package name or a unique namespace identifier that references their location within a package.

You compile packaged components similarly to how you compile components that are not in packages. The only difference is that you must use the package name in the namespace declaration, regardless of how you compiled the SWC file, and that package name uses dot-notation instead of slashes. You must be sure to specify the location of the classes in the `source-path`.

In the following command-line example, the MyButton component is in the mypackage package:

```
compc -source-path . c:/flexdeploy/comps/mypackage/
    -output c:/jrun4/servers/flex/WEB-INF/flex/user_classes/MyButtonComp.swc
    -include-classes mypackage.MyButton
```

These options appear in a configuration file, as the following example shows:

```
<compiler>
    <source-path>
        <path-element>.</path-element>
        <path-element>c:/flexdeploy/comps/mypackage/</path-element>
    </source-path>
    <output>
        c:/jrun4/servers/flex/WEB-INF/flex/user_classes/MyButtonComp.swc
    </output>
</compiler>
<include-classes>
    <class>mypackage.MyButton</class>
<include-classes>
```

To access the MyButton class in your application, you must declare a namespace that includes its package; for example:

```
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:mine="mypackage.*">
```

You can use the compc compiler to compile components from multiple packages into a single SWC file. In the following command-line example, the MyButton control is in the mypackage package, and the CustomComboBox control is in the acme package:

```
compc -source-path .
    -output c:/jrun4/servers/flex/WEB-INF/flex/user_classes/CustomComps.swc
    -include-classes mypackage.MyButton
    acme.CustomComboBox
```

You then define each package as a separate namespace in your MXML application:

```
<?xml version="1.0"?>
<s:Application <s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:mine="mypackage.*"
    xmlns:acme="acme.*">
    <mine:MyButton/>
    <acme:CustomComboBox/>
</s:Application>
```

Compiling components using namespaces

When you have many components in one or more packages that you want to add to a SWC file and want to reference from an MXML file through a custom namespace, you can list them in a manifest file, then reference that manifest file on the command line. Also, you can specify a namespace for that component or define multiple manifest files and, therefore, specify multiple namespaces to compile into a single SWC file.

When you use manifest files to define the components in your SWC file, you specify the namespace that the components use in your applications. You can compile all the components from one or more packages into a single SWC file. If you have more than one package, you can set it up so that all packages use a single namespace or so that each package has an individual namespace.

Components in a single namespace

In the manifest file, you define which components are in a namespace. The following sample manifest file defines two components to be included in the namespace:

```
<?xml version="1.0"?>
<!-- SimpleManifest.xml -->
<componentPackage>
    <component id="MyButton" class="MyButton"/>
    <component id="MyOtherButton" class="MyOtherButton"/>
</componentPackage>
```

The manifest file can contain references to any number of components in a namespace. The `class` option is the full class name (including package) of the class. The `id` property is optional, but you can use it to define the MXML tag interface that you use in your applications. If the compiler cannot find one or more files listed in the manifest, it throws an error. For more information on using manifest files, see “About manifest files” on page 2206.

On the command line, you define the namespace with the `namespace` option; for example:

```
-namespace http://mynamespace SimpleManifest.xml
```

Next, you target the defined namespace for inclusion in the SWC file with the `include-namespaces` option; for example:

```
-include-namespaces http://mynamespace
```

The `namespace` option matches a namespace (such as “http://ns.adobe.com/mxml/2009”) with a manifest file. The `include-namespaces` option instructs compc to include all the components listed in that namespace’s manifest file in the SWC file.

After you define the manifest file, you can compile the SWC file. The following command-line example compiles the components into the “http://mynamespace” namespace:

```
compc -source-path .
    -output c:/jrun4/servers/flex/WEB-INF/flex/user_classes/MyButtons.swc
    -namespace http://mynamespace SimpleManifest.xml
    -include-namespaces http://mynamespace
```

In a configuration file, these options appear as the following example shows:

```
<compiler>
    <source-path>
        <path-element>.</path-element>
    </source-path>
    <output>c:/jrun4/servers/flex/WEB-INF/flex/user_classes/MyButtons.swc</output>
    <namespaces>
        <namespace>
            <uri>http://mynamespace</uri>
            <manifest>SimpleManifest.xml</manifest>
        </namespace>
    </namespaces>
</compiler>
<include-namespaces>
    <uri>http://mynamespace</uri>
<include-namespaces>
```

In your application, you can access the components by defining the new namespace in the `<s:Application>` tag, as the following example shows:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:a="http://mynamespace">
    <a:MyButton/>
    <a:MyOtherButton/>
</s:Application>
```

Components in multiple namespaces

You can use the compc compiler to compile components that use multiple namespaces into a SWC file. Each namespace must have its own manifest file.

The following command-line example compiles components defined in the AcmeManifest.xml and SimpleManifest.xml manifest files:

```
compc -source-path .
    -output c:/jrun4/servers/flex/WEB-INF/flex/user_classes/MyButtons.swc
    -namespace http://acme2009 AcmeManifest.xml
    -namespace http://mynamespace SimpleManifest.xml
    -include-namespaces http://acme2009 http://mynamespace
```

In this case, all components in both the http://mynamespace and http://acme2009 namespaces are targeted and included in the output SWC file.

In a configuration file, these options appear as the following example shows:

```
<compiler>
    <source-path>
        <path-element>.</path-element>
    </source-path>
    <output>c:/jrun4/servers/flex/flex/WEB-INF/flex/user_classes/MyButtons.swc</output>
    <namespaces>
        <namespace>
            <uri>http://acme2009</uri>
            <manifest>AcmeManifest.xml</manifest>
        </namespace>
        <namespace>
            <uri>http://mynamespace</uri>
            <manifest>SimpleManifest.xml</manifest>
        </namespace>
    </namespaces>
</compiler>
<include-namespaces>
    <uri>http://acme2009</uri>
    <uri>http://mynamespace</uri>
<include-namespaces>
```

In your MXML application, you define both namespaces separately:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:simple="http://mynamespace"
    xmlns:acme="http://acme2009">
    <simple:SimpleComponent/>
    <acme:AcmeComponent/>
</s:Application>
```

You are not required to include all namespaces that you define as target namespaces. You can define multiple namespaces, but use only one target namespace. You might do this if some components use other components that are not directly exposed as MXML tags. You cannot then directly access the components in the unused namespace, however.

The following command line example defines two namespaces, http://acme2009 and http://mynamespace, but only includes one as a namespace target:

```
compc -source-path .
    -output c:/jrun4/servers/flex/flex/WEB-INF/flex/user_classes/MyButtons.swc
    -namespace http://acme2009 AcmeManifest.xml
    -namespace http://mynamespace SimpleManifest.xml
    -include-namespaces http://mynamespace
```

Adding utility classes

You can add any classes that you want to use in your applications to a SWC file. These classes do not have to be components, but are often files that components use. They are classes that might be used at run time and, therefore, are not checked by the compiler. For example, your components might use a library of classes that perform mathematical functions, or use a custom logging utility. This documentation refers to these classes as *utility classes*. Utility classes are not exposed as MXML tags.

To add utility classes to a SWC file, you use the `include-sources` option. This option lets you specify a path to a class file rather than the class name, or specify an entire directory of classes.

The following command-line example adds the FV_calc.as and FV_format.as utility classes to the SWC file:

```
compc -source-path .
    -output c:/jrun4/servers/flex/WEB-INF/flex/user_classes/MySwc.swc
    -include-sources FV_classes/FV_format.as FV_classes/FV_calc.as
    -include-classes asbutton.MyButton
```

In a configuration file, these options appear as the following example shows:

```
<compiler>
    <source-path>
        <path-element>.</path-element>
    </source-path>
    <output>c:/jrun4/servers/flex/flex/WEB-INF/flex/user_classes/MySwc.swc</output>
</compiler>
<include-classes>
    <class>asbutton.MyButton</class>
</include-classes>
<include-sources>
    <path-element>FV_classes/FV_format.as</path-element>
    <path-element>FV_classes/FV_calc.as</path-element>
<include-sources>
```

When specifying files with the `include-sources` option, you must give the full filename (for example, FV_calc.as instead of FV_calc) because the file is not a component. If you use this option to include MXML components that are in a non-default package, you must include the source folder in the source path.

You can also provide a directory name to the `include-sources` option. In this case, the compiler includes all files with an MXML or AS extension, and ignores all other files.

Classes that you add with the `include-sources` option can be accessed from the generic namespace in your applications. To use them, you need to add the following code in your application tag:

```
xmlns:local="*"
```

You can then use them as tags; for example:

```
<local:FV_calc id="calc" rate=".0125" nper="12" pmt="100" pv="0" type="1"/>
```

Adding nonsource classes

You often include noncompiled (or nonsource) files with your applications. A *nonsource* file is a class or resource (such as a style sheet or graphic) that is not compiled but is included in the SWC file for other classes to use. For example, a font file that you embed or a set of images that you use as graphical skins in a component's style sheet should not be compiled but should be included in the SWC file. These are classes that you typically do not use the `[Embed]` syntax to link in to your application.

Use the `include-file` option to define nonsource files in a SWC file.

The syntax for the `include-file` option is as follows:

```
-include-file name path
```

The *name* argument is the name used to reference the embedded file in your applications. The *path* argument is the current path to the file in the file system.

When you use the `include-file` option, you specify both a name and a filepath, as the following example shows:

```
compc -include-file logo.gif c:/images/logo/logo1.gif ...
```

In a configuration file, these options appear as the following example shows:

```
<compiler>
    <output>c:/jrun4/servers/flex/flex/WEB-INF/flex/user_classes/Combo.swc</output>
</compiler>
<include-file>
    <name>logo.gif</name>
    <path>c:/images/logo/logo1.gif</path>
</include-file>
<include-classes>
    <class>asbutton.MyButton</class>
<include-classes>
```

Each name that you assign to a resource must be unique because the name becomes a global variable.

You cannot specify a list of files with the `include-file` option. So, you must add a separate `include-file` option for each file that you include, as the following command-line example shows:

```
compc -include-file file1.jpg ../images/file1.jpg -include-file file2.jpg ../images/file2.jpg
-- -output MyFile.swc
```

If you want to add many resources to the SWC file, consider using a configuration file rather than listing all the resources on the command line. For an example of a configuration file that includes multiple resources in a SWC file, see “Styles and themes” on page 1492.

In general, specify a file extension for files that you include with the `include-file` option. In some cases, omitting the file extension can lead to a loss of functionality. For example, if you include a CSS file in a theme SWC file, you must set the name to be *.css. When Flex examines the SWC file, it applies all CSS files in that SWC file to the application. CSS files without the CSS extension are ignored.

Creating themes

You can use the `include-file` and `include-classes` options to add skin files and style sheets to a SWC file. The SWC file can then be used as a theme. For more information about using themes in applications, see “Styles and themes” on page 1492.

Viewing errors and warnings

You can use the compiler options to specify what level of warnings and errors to view. Also, you can set levels of logging with the compiler options.

Viewing warnings and errors

There are several options that let you customize the level of warnings and errors that are displayed by the Flex compilers, including the following:

- `show-actionscript-warnings`
- `show-binding-warnings`
- `show-invalid-css-property-warnings`
- `show-shadowed-device-font-warnings`
- `show-unused-type-selector-warnings`
- `strict`
- `warnings`

To disable all warnings, set the `warnings` option to `false`.

The `show-actionscript-warnings` option displays compiler warnings for the following situations:

1. Situations that are probably not what the developer intended, but are still legal; for example:

   ```
   if (a = 10)                 // Did you really want '==' instead of '='?
   if (b == NaN)               // Any comparison with NaN is always false.
   var b;                      // Missing type declaration.
   ```

2. Usage of deprecated or removed ActionScript 2.0 APIs.
3. Situations where APIs behave differently in ActionScript 2.0 than in ActionScript 3.0.

You can customize the types of warnings displayed by using options that begin with *warn* (for example, `warn-constructor-return-values` and `warn-bad-type-cast`). A complete list of warnings are available in the advanced command-line help or in the flex-config.xml file.

The `strict` option enforces typing and reports run-time verifier errors at compile time. This option assumes that definitions are not dynamically redefined at run time, so these checks can be made at compile time. It displays errors for conditions such as undefined references, const and private violations, argument mismatches, and type checking.

The `show-binding-warnings` option displays warnings when Flash Player cannot detect changes to bound properties.

About deprecation

In some cases, Flex functionality has been deprecated. Deprecated features and properties have the following characteristics:

- Generate compiler warnings.
- Continue to work in Flex 4.x.
- Will be removed from the product in a future major release.

The command-line compilers report deprecation warnings by default. You can suppress deprecation warnings by setting the `show-deprecation-warnings` option to `false`.

In some cases, deprecated functionality does not work as expected unless you instruct the compiler to use an earlier version of the compiler with the `compatibility-version` compiler argument. For example, to use functionality that was deprecated since Flex 3, you could use the following command-line compiler argument:

```
-compatibility-version=3.0.0
```

Possible values for this compiler argument are defined as constants in the FlexVersion class. For more information on using this compiler argument, see “Backward compatibility” on page 2239.

You can use the `[Deprecated]` metadata tag to deprecate your own classes and class elements. For more information, see “Deprecated metadata tag” on page 2383.

About logging

Errors and warnings are reported differently, depending on which compiler you are using.

The mxmlc and compc command-line compilers send error and warning messages to the standard output. You can redirect this output by using the redirector (>).

Flash Builder displays error and warning messages in the Problems tab.

About SWC files

A *SWC* file is an archive file, sometimes also referred to as a class library, for components and other assets. SWC files contain a SWF file and a catalog.xml file, in addition to properties files and other uncompiled assets such as CSS files. The SWF file implements the compiled component or group of components and includes embedded resources as symbols. An application extracts the SWF file from a SWC file. It uses the SWF file's contents when the application refers to resources in that SWC file. The catalog.xml file lists of the contents of the component package and its individual components.

In most cases, the symbols defined in the SWF file that are referenced by the application are embedded in the application at compile-time. This is known as static linking. The application compiler only includes those classes that are used by your application, and dependent classes, in the final SWF file.

You can also dynamically link the contents of SWC files. Dynamic linking is when the entire SWF file is loaded at run time. To achieve dynamic linking of the SWF file, you must use the SWC file as a Runtime Shared Library, or RSL. For more information, see "Runtime Shared Libraries" on page 253.

SWC files make it easy to exchange components and other assets among Flex developers. You need only exchange a single file, rather than the MXML or ActionScript files and images and other resource files. The SWF file in a SWC file is compiled, which means that the code is loaded efficiently and it is hidden from casual view. Also, compiling a component as a SWC file can make namespace allocation an easier process.

You can package and expand SWC files with tools that support the PKZip archive format, such as WinZip or jar. However, do not manually change the contents of a SWC file, and do not try to run the SWF file that is in a SWC file in Flash Player.

You typically create SWC files by using compc, the command-line component compiler.

You can also save SWC files as open directories rather than archived files. This gives you easier access to the contents of the SWC file. You create an open directory SWC by setting the `directory` option to `true`. This is typically only used when you create a custom RSL because RSLs require that you deploy the SWF file with your application.

The properties files inside SWC files are uncompiled text files in UTF-8 format. They are used as resource bundles for localization of applications. Within the SWC file, they are stored in the /locale/*locale_string* directory, where `locale_string` is something like en_US or fr_FR. For more information, see "Localization" on page 2055.

Distributing SWC files

After you generate a SWC file, you can use it in your applications.

You can also copy SWC files to a directory specified by the `library-path` compiler option. You must store SWC files at the top level of the user_classes directory or the directory specified by the `library-path`. You cannot store SWC files in subdirectories.

To use a SWC file when compiling components or applications from the command line or from within Flash Builder, you specify the location of the SWC file with the `library-path` compiler option.

***Note:** Do not store custom components or classes in the flex_root/WEB-INF/flex/libs directory. This directory is for Adobe classes and components.*

Using components in SWC files

If a component in a SWC file does not have a namespace, you can add a generic namespace identifier in your `<s:Application>` tag to use the component, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:a="*">
```

If the component has a package name as part of its namespace, you must do one of the following:

1 Add the package name to the namespace declaration in the `<s:Application>` tag; for example:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:but="mycomponents.*">
```

2 Create a manifest file and recompile the SWC file. You pass the manifest file to the compc compiler by using the `namespace` option. In the `<s:Application>` tag, you specify only the unique namespace URI that you used with compc. For more information on specifying a namespace for the component, see “Compiling components using namespaces” on page 2199.

About manifest files

Manifest files map a component namespace to class names. They define the package names that the components used before being compiled into a SWC file. They are not required when compiling SWC files, but they can help keep your source files organized.

Manifest files use the following syntax:

```
<?xml version="1.0"?>
<componentPackage>
    <component id="component_name" class="component_class"/>
    [...]
</componentPackage>
```

For example:

```
<?xml version="1.0"?>
<componentPackage>
    <component id="MyButton" class="package1.MyButton"/>
    <component id="MyOtherButton" class="package2.MyOtherButton"/>
</componentPackage>
```

In a manifest file, the `id` property of each `<component>` tag must be unique. It is the name you use for the tag in your applications. For example, you define the `id` as `MyButton` in the manifest file:

```
<component id="MyButton" class="asbutton.MyButton"/>
```

In your application, you use `MyButton` as the tag name:

```
<local:MyButton label="Click Me"/>
```

The `id` property in the manifest file entry is optional. If you omit it, you can use the class name as the tag. This is useful if you have two classes with the same name in different packages. In this case, you use the manifest to define the tags, as the following example shows:

```
<?xml version="1.0"?>
<componentPackage>
    <component id="BoringButton" class="boring.MyButton"/>
    <component id="GreatButton" class="great.MyButton"/>
</componentPackage>
```

Some SWC files consist of multiple components from different packages, so compc includes a manifest file with your SWC file in those cases to prevent compiler errors.

When compiling the SWC file, you specify the manifest file by using the `namespace` and the `include-namespaces` options. You define the namespace and its contents with the namespace option:

```
-namespace http://mynamespace mymanifest.xml
```

Then you identify that namespace's contents for inclusion in the SWC file:

```
-include-namespaces http://mynamespace
```

Using fcsh, the Flex compiler shell

The fcsh (Flex Compiler Shell) utility provides a shell environment that you use to compile applications, modules, and component libraries. It works very similarly to the mxmlc and compc command line compilers, but it compiles faster than the mxmlc and compc command-line compilers. One reason is that by keeping everything in memory, fcsh eliminates the overhead of launching the JVM and loading the compiler classes. Another reason is that compilation results (for example, type information) can be kept in memory for subsequent compilations.

This utility is intended to improve the experience of users of the command-line compilers. If you are using Flash Builder, you do not need to use fcsh. The Flash Builder tool already uses the optimizations provided by fcsh.

For simple applications, fcsh might not be necessary. But for more complex applications that you compile frequently, you should experience a significant performance improvement over using the mxmlc and compc command-line compilers.

When you first compile an application with fcsh, you will not typically notice any difference in speed between the fcsh and the command-line compilers. This is because fcsh must load the application model and custom libraries into memory, just as the command-line compilers would. After that, however, each subsequent compilation uses the libraries in memory to compile. This reduces the amount of disk access that the compilers need to perform and should result in shorter compile times.

The fcsh tool is in the bin directory. For Unix and Mac OS, it is a shell script called fcsh. For Windows, it is fcsh.exe. You invoke it only from the command line. The Java settings are managed by the jvm.config file in the bin directory.

Using fcsh

You invoke fcsh from the command line. You can launch the utility either as an executable (Windows) or shell command (Unix/Linux/Mac).

1 Open a command prompt.

2 Navigate to the {*SDK_root*}/bin directory.

3 Enter `fcsh` at the command line. Your commands will now be executed within the fcsh environment. You will know this if your command prompt becomes `(fcsh)`.

Typically, you compile a simple application when you first launch fcsh; for example:

```
(fcsh) mxmlc c:/myfiles/MyApp.mxml
```

The fcsh utility returns a target id:

```
fcsh: Assigned 1 as the compile target id.
```

You can then refer to the target ids when using subsequent commands inside the fcsh utility. For example, to compile the previous application with incremental compilation:

```
(fcsh) compile 1
```

You can enter `help` in the fcsh shell to see a list of available options; for example:

```
(fcsh) help
```

You can enter `quit` in the fcsh shell to exit fcsh and return to the command prompt; for example:

```
(fcsh) quit
```

The following example shows that fcsh dramatically reduces compilation time when compiling the same application multiple times. The first compilation takes 8885 milliseconds. The second takes only 5140 milliseconds:

```
(fcsh) mxmlc -benchmark=true flexstore.mxml
    Total time: 8885ms
    Peak memory usage: 84 MB (Heap: 58, Non-Heap: 26)
(fcsh) mxmlc -benchmark=true flexstore.mxml
    Total time: 5140ms
    Peak memory usage: 84 MB (Heap: 57, Non-Heap: 27)
```

In addition, subsequent *full* compilations of the same application are much faster, as the following example shows:

```
> touch flexstore.mxml

(fcsh) compile 1
    Files changed: 1 Files affected: 0
    Total time: 933ms
    Peak memory usage: 88 MB (Heap: 62, Non-Heap: 26)
    flexstore.swf (522456 bytes)
    Total time: 1102ms
    Peak memory usage: 77 MB (Heap: 51, Non-Heap: 26)
(fcsh)
```

About fcsh commands

The following table describes the available fcsh commands:

Option	Description
`clear [`*id*`]`	Removes the target id(s) from memory but saves the target's *.cache file. If you enter this command without specifying an id argument, fcsh clears all target ids. For information about cache files, see “About incremental compilation” on page 2191.
`compile` *id*	Uses incremental compilation (without linking) to compile the specified id. This command uses the same arguments used in previous compilations. If you try to compile a target that has not changed, fcsh skips that target.
`compc` *arg1* `[...]`	Compiles SWC files from the specified sources. This command returns a target id that you can then pass to other fcsh options. The target ids are incremented by 1 for each new compilation. If you quit fcsh and then relaunch it, all targets are cleared and the ids start at 1 again. This command takes all the same arguments as the compc command-line compiler. For information about using compc, see “Using compc, the component compiler” on page 2194.
`help`	Lists commands. Commands not listed but also supported include `cp`, `mv`, `rm`, and `touch`.

Option	Description
`info [id]`	Displays compiler target information such as the source files and cache file name. If you do not specify a target id, fcsh prints information for all targets in reverse id order.
`mxmlc arg1 [...]`	Compiles and optimizes the target application or module using the mxmlc command-line compiler. This command returns a target id that you can then pass to other fcsh options. The target ids are incremented by 1 for each new compilation. If you quit fcsh and then relaunch it, all targets are cleared and the ids start at 1 again. This command takes all the same arguments as the mxmlc command-line compiler. For information about using mxmlc, see "Using mxmlc, the application compiler" on page 2174.
`quit`	Exits the fcsh utility. All data stored in memory is destroyed. When you relaunch fcsh, you cannot access targets that you created in a previous session.

The fcsh tool also supports the following commands:

- `cp`
- `mv`
- `rm`
- `touch`

Command-line debugger

If you encounter errors in your applications, you can use the debugging tools to set and manage breakpoints in your code; control application execution by suspending, resuming, and terminating the application; step into and over the code statements; select critical variables to watch; evaluate watch expressions while the application is running, and so on.

About debugging

Debugging Adobe® Flex® applications can be as simple as enabling `trace()` statements or as complex as stepping into an application's source files and running the code, one line at a time. The Adobe® Flash® Builder™ debugger and the command-line debugger, fdb, let you step through and debug the files used by your Flex applications.

To debug a Flex application, you must generate a debug SWF file. This is a SWF file with debug information in it. You then connect fdb to the debugger version of Adobe® Flash® Player that is running the debug SWF file.

The debugger is an agent that communicates with the application that is running in Flash Player. It connects to your application with a local socket connection. As a result, you might have to disable anti-virus software to use it if your anti-virus software prevents socket communication. The debugger uses this connection to transfer information from the SWF file to the command line so that you can add breakpoints, inspect variables, and do other common debugging tasks. The port through which the debugger connects to your application is 7935. You cannot change this port.

This topic describes how to use the fdb command-line debugger. To use the Flash Builder debugger, see Debugging Tools in Flash Builder.

To use either debugger, you must install and configure the debugger version of Flash Player. To determine if you are running the debugger version or the standard version of Flash Player, open any Flex application in the player and right-click the mouse button. If you see the Show Redraw Regions option, you are running the debugger version of Flash Player. For more information about the debugger version of Flash Player, and how to detect which player you are running, see "Using the debugger version of Flash Player" on page 2223.

Using the command-line debugger

The fdb command-line debugger is located in the *flex_install_dir*/bin directory. To start fdb, open a command prompt, change to that directory, and enter **fdb**.

For a description of available commands, use the following command:

```
(fdb) help
```

For an overview of the fdb debugger, use the following command:

```
(fdb) tutorial
```

Generating debug SWF files

To debug a Flex application, you first generate a debug SWF file. Debug SWF files are similar to other application SWF files except that they contain debugging-specific information that the debugger and the debugger version of Flash Player use during debugging sessions. Debug SWF files are larger than non-debug SWF files, so generate them only when you are going to debug with them.

To generate the debug SWF file using the mxmlc command-line compiler, you set the `debug` option to `true`, either on the command line or in the flex-config.xml file. The following example sets the `debug` option to `true` on the command line:

```
mxmlc -debug=true myApp.mxml
```

Flash Builder generates debug SWF files by default in the project's /bin-debug directory. To generate a non-debug SWF file in Flash Builder, you use the Export Release Build feature. This generates a non-debug SWF file in the project's /bin-release directory. You can also manually set the `debug` compiler option to `false` in the Additional Compiler Arguments field of the project's properties.

You can use fdb in any gdb-compatible debugging environment. For example, you can use M-x gdb inside Emacs and specify fdb.exe as the gdb command.

Command-line debugger limitations

The command-line debugger supports debugging only at the ActionScript level and does not support the Flash timeline concept. The debugger also does not support adding breakpoints inside script snippets in MXML tags. You can set breakpoints on event handlers defined for MXML tags.

Flash Player may interact with a server. The debugger does not assist in debugging the server-side portion of the application, nor does it offer support for inspecting any of the IP transactions that take place from Flash Player to the server, and vice versa.

Command-line debugger shortcuts

You can open commands within the fdb debugger by using the fewest number of nonambiguous keystrokes. For example, to use the `print` command, you can type **p**, because no other command begins with that letter.

Using the default browser

When you debug an application in a web browser, fdb opens the player in the default browser. The *default browser* is the browser that opens when you open a web-specific file without specifying an application. You must also have the debugger version of Flash Player installed with this browser. If you do not have the debugger version of Flash Player, Flash displays an error indicating that your Flash Player does not support all fdb commands.

Your default browser might not be the first browser that you installed on your computer. For example, if you installed another web browser *after* installing Microsoft Internet Explorer, Internet Explorer might not be your default browser.

Determine your default browser

1 From the Windows toolbar, select Start.

2 Select Run and enter a URL in the Run dialog box. For example:

```
http://www.adobe.com
```

3 Click OK.

Windows opens the default browser or displays an error message indicating that there is no application configured to handle your request.

Set Internet Explorer 6.x as your default browser

1 Open the Internet Explorer application.

2 Select Tools > Internet Options.

3 Select the Programs tab.

4 Select the "Internet Explorer should check to see whether it is the default browser" option, and click OK.

The next time you start Internet Explorer, Internet Explorer prompts you to make it the default browser. If you are not prompted, Internet Explorer is already your default browser.

Set Firefox as your default browser

1 Open the Firefox application.

2 Select Tools > Options.

3 Select the General icon to view general settings.

4 Select the "Firefox should check to see if it is the default browser when starting" option, and click OK.

The next time you start FireFox, FireFox prompts you to make it the default browser. If you are not prompted, FireFox is already your default browser.

About the source files

Each application can have any number of ActionScript files. Some of the files that fdb steps into are external class files, and some are generated by the Flex compilers.

In general, Flex generates a single file that contains ActionScript statements used in `<fx:Script>` blocks in the root MXML file, and an additional file for each ActionScript class that the application uses. Flex generates many source files so that you can navigate the application from within the debugger.

To view a list of files that are used by the application you are debugging, use the `info files` command. For more information, see "Getting status" on page 2220.

The generated ActionScript class files are sometimes referred to as compilation units. For more information about compilation units, see "About incremental compilation" on page 2191.

Using RSLs when debugging

By default, you compile applications against the signed framework RSLs. The signed framework RSLs are optimized, which means that they do not include debugging information. Flash Builder automatically uses unoptimized debug RSLs when you launch the debugger. If you compile on the command line, however, and use the command line debugger, you must either disable RSLs or specify non-optimized RSLs to use. Otherwise, you will not be able to set break points or take advantage of other debugging functionality.

For more information, see "Example of using the framework RSLs on the command line" on page 266

Starting a debugging session

You start a debugging session by using the fdb command-line debugger. After you start a session, you typically type **continue** once before you set break points and perform other debugging tasks. This is because the first frame that suspends debugging occurs before the application has finished initialization.

For more information about which commands are available after you start a debugging session, see “Using the command-line debugger commands” on page 2213.

Starting a session with the stand-alone Flash Player

You can start a debugging session with the stand-alone debugger version of Flash Player. You do this by compiling the application into a SWF file, and then invoking the SWF file with the fdb command-line debugger. The fdb debugger opens the debugger version of the stand-alone Flash Player.

The debugger version of the stand-alone Flash Player runs as an independent application. It does not run within a web browser or other shell. The debugger version of the stand-alone Flash Player does not support any server requests, such as web services and dynamic SWF loading, so not all applications can be properly debugged inside the debugger version of the stand-alone Flash Player.

Debug with the stand-alone Flash Player

1 Compile the Flex application’s debug SWF file and set the `debug` option to `true`.

The following example compiles an application with the mxmlc command-line compiler:

```
mxmlc -debug=true myApp.mxml
```

You can also compile an application SWF file by using the Flash Builder compiler. For more information on Flex compilers, see “About the Flex compilers” on page 2164.

2 Find the *flex_install_dir*/bin directory. You installed the Flex application files to this directory.

3 Type **fdb** from the command line. The fdb prompt appears.

You can also open fdb with the JAR file, as the following example shows:

```
 java -jar ../lib/fdb.jar
```

4 Type **run** at the fdb prompt, followed by the path to the SWF file, as shown in the following example:

```
(fdb) run c:/myfiles/fonts/EmbedMyFont.swf
```

The fdb debugger starts the Flex application in the debugger version of the stand-alone Flash Player, and the `(fdb)` command prompt appears. You can also start a session by typing **fdb** *filename***.swf** at the command prompt, rather than by using the `run` command.

Starting a session in a browser

You can start a debugging session in a browser. This requires that you pre-compile the SWF file and are able to request it from a web server.

Debug in a browser

1 Compile the Flex application’s debug SWF file and set the `debug` option to `true`. The following example compiles an application with the mxmlc command-line compiler:

```
mxmlc -debug=true myApp.mxml
```

You can also compile an application SWF file by using the Flash Builder compiler. For more information on Flex compilers, see “About the Flex compilers” on page 2164.

2 Create an HTML wrapper that embeds this SWF file, if you have not already done so. For more information on creating a wrapper, see "Creating a wrapper" on page 2552.

3 Copy the SWF file and its wrapper files to your web server.

4 Find the *flex_install_dir*/bin directory. You installed the Flex application files to this directory.

5 Type **fdb** in the command line. The fdb prompt appears.

You can also open fdb with the JAR file, as the following example shows:

```
java -jar ../lib/fdb.jar
```

6 Type **run** at the fdb prompt:

```
(fdb) run
```

This instructs fdb to wait for a player to connect to it.

7 In your browser, request a wrapper that embeds the debug SWF file. Do not request the SWF file directly in a browser because some browsers do not allow you to run a SWF file directly.

Alternatively, you can type **run** *filename*.**html** at the command line, and fdb launches the browser for you. The filename should include the entire URL; for example:

```
(fdb) run http://localhost:8100/flexapps/index.html
```

Configuring the command-line debugger

You can configure the current session of the fdb command-line debugger using variables that exist entirely within fdb; they are not part of your application. The configuration variables are prefixed with $.

The following table describes the most common configuration variables used by fdb:

Variable	Description
`$invokegetters`	Set to 0 to prevent fdb from firing getter functions. The default value is `1` (enabled).
`$listsize`	Sets the number of lines to display with the list command. The default value is `10`.

To set the value of a configuration variable, you use the `set` command, as the following example shows:

```
(fdb) set $invokegetters = 0
```

For more information on using the `set` command, see "Changing data values" on page 2218.

Using the command-line debugger commands

The fdb command-line debugger includes commands that you use to debug and navigate your Flex application.

Running the debugger

The fdb debugger provides several commands for stepping through the debugged application's files. The following table summarizes those commands:

Command	**Description**
`continue`	Continues running the application.
`file [`*`file`*`]`	Specifies an application to be debugged, without starting it. This command does not cause the application to start; use the `run` command without an argument to start debugging the application.
`finish`	Continues until the function exits.
`next [`*`N`*`]`	Continues to the next source line in the application. The optional argument ***N*** means do this ***N*** times or until the program stops for some other reason.
`quit`	Exits from the debug session.
`run [`*`file`*`]`	Starts a debugging session by running the specified file. To run the application that the file command previously specified, execute the `run` command without any options. The `run` command starts the application in a browser or stand-alone Flash Player.
`step [`*`N`*`]`	Steps into the application. The optional argument ***N*** means do this ***N*** times or until the program stops for some other reason. These commands are nonblocking, which means that when they return, the client has at least begun the operation, but it has not necessarily finished it.

When you start a session, fdb stops execution before Flex renders the application on the screen. Use the `continue` command to get to the application's starting screen.

The following example shows a sample application after it starts:

```
(fdb) continue
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] RadioButtonGroup.addInstance: instance = _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton1 data = undefined label = 2005
[trace] RadioButtonGroup.addInstance: instance = _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton2 data = undefined label = 2004
[trace] RadioButtonGroup.addInstance: instance = _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton3 data = undefined label = 2005
[trace] RadioButtonGroup.addInstance: instance = _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton4 data = undefined label = 2006
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 14
```

During the debugging session, you interact with the application in the debugger version of Flash Player. For example, if you select an item from the drop-down list, the debugger continues to output information to the command window:

```
[trace] SSL : ConfigureScrolling
[trace] SSP : 5 51 true 47
[trace] ComboBase: y = 0 text_mc.bl = 14
[trace] layoutChildren : bRowHeightChanged
[trace] >>SSL:layoutChildren
[trace] deltaRows 5
[trace] rowCount 5
[trace] <<SSL:layoutChildren
[trace] >>SSL:draw
[trace] bScrollChanged
[trace] SSL : ConfigureScrolling
[trace] SSP : 5 51 false 46
[trace] SSL Drawing Rows in UpdateControl 5
[trace] <<SSL:draw
```

You can store commonly used commands in a source file, and then load that file by using the `source` command. For more information, see “Accessing commands from a file” on page 2217.

Setting breakpoints

Setting breakpoints is a critical aspect of debugging any application. You can set breakpoints on any ActionScript code in your Flex application. You can set breakpoints on statements in any external ActionScript file, on ActionScript statements in an `<fx:Script>` tag, or on MXML tags that have event handler properties. In the following MXML code, `click` is an event handler property:

```
<s:Button click="ws.getWeather.send();"/>
```

Breakpoints are maintained from session to session. However, when you change the target file or quit fdb, breakpoints are lost.

The following table summarizes the commands for manipulating breakpoints with the ActionScript debugger:

Command	Description
`break [`***`args`***`]`	Sets a breakpoint at the specified line or function. The argument can be a line number or function name. With no arguments, the `break` command sets a breakpoint at the currently stopped line (not the currently listed line). If you specify a line number, fdb breaks at the start of code for that line. If you specify a function name, fdb breaks at the start of code for that function.
`clear [`***`args`***`]`	Clears a breakpoint at the specified line or function. The argument can be a line number or function name. If you specify a line number, fdb clears a breakpoint in that line. If you specify a function name, fdb clears a breakpoint at the beginning of that function. With no argument, fdb clears a breakpoint in the line that the selected frame is executing in. Compare the `delete` command, which clears breakpoints by number.
`commands [`*`breakpoint`*`]`	Sets commands to execute when the specified breakpoint is encountered. If you do not specify a breakpoint, the commands are applied to the last breakpoint.

Command	Description

`condition` *bp_num* [*expression*]	Specifies a condition that must be met to stop at the given breakpoint (identified by the breakpoint number). The fdb debugger evaluates expression when the breakpoint is reached. If the value is `true` or nonzero, fdb stops at the breakpoint. Otherwise, fdb ignores the breakpoint and continues execution.
	To remove the condition from the breakpoint, do not specify an expression.
	You can use conditional breakpoints to stop on all events of a particular type. For example, to stop on every `initialize` event, use the following commands:
	`(fdb) ` **`break UIEvent:dispatch`**
	`Breakpoint 18 at 0x16cb3: file UIEventDispatcher.as, line 190(fdb)`
	`condition 18 (eventObj.type == 'initialize')`

`delete` [***args***]	Deletes breakpoints. Specify one or more comma- or space-separated breakpoint numbers to delete those breakpoints. To delete all breakpoints, do not provide an argument.
`disable breakpoints` [*bp_num*]	Disables breakpoints. Specify one or more space-separated numbers as options to disable only those breakpoints.
`enable breakpoints` [*bp_num*]	Enables breakpoints that were previously disabled. Specify one or more space-separated numbers as options to enable only those breakpoints.

The following example sets a breakpoint on the `myFunc()` method, which is triggered when the user clicks a button:

```
(fdb) break myFunc
Breakpoint 1 at 0x401ef: file file1.mxml, line 5
(fdb) continue
Breakpoint 1, myFunc() at file1.mxml:5
    5ta1.text = "Clicked";
(fdb)
```

To see all breakpoints and their numbers, use the `info breakpoints` command. This will also tell you if a breakpoint is unresolved.

You can use the `commands` command to periodically print out values of objects and variables whenever fdb encounters a particular breakpoint. The following example prints out the value of ta1.text (referred to as `$1`), executes the `where` command, and then continues when it encounters the button's click handler breakpoint:

```
(fdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just 'end'.
>print ta1.text
>where
>continue
>end
(fdb) cont
Breakpoint 1, myFunc() at file1.mxml:5
    5ta1.text = "Clicked";
$1 = ""
#0  [MovieClip 1].myFunc(event=undefined) at file1.mxml:5
#1  [MovieClip 1].handler(event=[Object 18127]) at file1.mxml:15
```

Breakpoints are not specific to a single SWF file. If you set a breakpoint in a file that is common to multiple SWF files, fdb applies the breakpoint to all SWF files.

For example, suppose you have four SWF files loaded and each of those SWF files contains the same version of an ActionScript file, view.as. To set a breakpoint in the `init()` function of the view.as file, you need to set only a single breakpoint in one of the view.as files. When fdb encounters any of the `init()` functions, it triggers the break.

By using the `watch` command, you can set watchpoints on variables. When that variable changes, the debugger halts. Watched variables can be any member variable, but cannot be dynamic, getters, or local variables. The `watch` command has the following syntax:

```
(fdb) watch variable_name
```

Accessing commands from a file

You can use the `source` command to read fdb commands from a file and execute them. This lets you write commands such as breakpoints once and use them repeatedly when debugging the same application in different sessions or across different applications.

The `source` command has the following syntax:

```
(fdb) source file
```

The value of file can be a filename for a file in the current working directory or an absolute path to a remote file. To determine the current working directory, use the `pwd` command.

The following examples read in the mycommands.txt file from different locations:

```
(fdb) source mycommands.txt
(fdb) source mydir\mycommands.txt
(fdb) source c:\mydir\mycommands.txt
```

Examining data values

The `print` command displays values of members such as variables, objects, and properties. This command excludes functions, static variables, constants, and inaccessible member variables (such as the members of an Array).

The `print` command uses the following syntax:

```
print [variable_name | object_name[.] | property]
```

The `print` command prints the value of the specified variable, object, or property. You can specify the `name` or `name.property` to narrow the results. If fdb can determine the type of the entity, fdb displays the type.

If you specify the `print` command on an object, fdb displays a numeric identifier for the object.

To list all the properties of an object, use trailing dot-notation syntax. The following example prints all the properties of the object myButton:

```
(fdb) print myButton.
```

To print the value of a single variable, use dot-notation syntax, as the following example shows:

```
(fdb) print myButton.label
```

Use the `what` command to view the context of a variable. The `what` command has the following syntax:

```
(fdb) what variable
```

Use the `display` command to add an expression to the autodisplay list. Every time debugging stops, fdb prints the list of expressions in the autodisplay list. The `display` command has the following syntax:

```
(fdb) display [expression]
```

The expression is the same as the arguments for the `print` command, as the following example shows:

```
(fdb) display myButton.color
```

To view all expressions on the autodisplay list, use the `infodisplay` command.

To remove an expression from the autodisplay list, use the `undisplay` command. The `undisplay` command has the following syntax:

```
(fdb) undisplay [list_num]
```

Use the `undisplay` command without an argument to remove all entries on the autodisplay list. Specify one or more `list_num` options separated by spaces to remove numbered entries from the autodisplay list.

You can temporarily disable autodisplay expressions by using the `disabledisplay` command. The `disable display` command has the following syntax:

```
(fdb) disable display [display_num]
```

Specify one or more space-separated numbers as options to disable only those entries in the autodisplay list.

To re-enable the display list, use the `enabledisplay` command, which has the same syntax as the `disable display` command.

Changing data values

You can use the `set` command to assign the value of a variable or a configuration variable. The `set` command has the following syntax:

```
set [expression]
```

Depending on the variable type, you use different syntax for the *expression*. The following example sets the variable `i` to the number 3:

```
(fdb) set i = 3
```

The following example sets the variable `employee.name` to the string `Reiner`:

```
(fdb) set employee.name = "Reiner"
```

The following example sets the convenience variable `$myVar` to the number `20`:

```
(fdb) set $myVar = 20
```

You use the `set` command to set the values of fdb configuration variables. For more information, see “Configuring the command-line debugger” on page 2213.

Viewing file contents

You use the `list` command to view lines of code in the ActionScript files. The `list` command uses the following syntax:

```
list [- | line_num[,line_num] | [file_name:]line_num | file_name[:line_num] |
[file_name:]function_name]
```

You use the `list` command to print the lines around the specified function or line of the current file. If you do not specify an argument, `list` prints 10 lines after or around the previous listing. If you specify a filename, but not a line number, `list` assumes line 1.

If you specify a single numeric argument, the `list` command lists 10 lines around that line. If you specify more than one comma-separated numeric argument, the `list` command displays lines between and including those line numbers.

To set the list location to where the execution is currently stopped, use the `home` command.

The following example lists code from line 10 to line 15:

```
(fdb) list 10, 15
```

If you specify a hyphen (-) in the previous example, the `list` command displays the 10 lines before a previous 10-line listing.

Specify a line number to list the lines around that line in the current file, as in the following example:

```
(fdb) list 10
```

Specify a filename followed by a line number to list the lines around that line in that file, as in the following example:

```
(fdb) list effects.mxml:10
```

Specify a function name to list the lines around the beginning of that function, as in the following example:

```
(fdb) list myFunction
```

Specify a filename followed by a function name to list the lines around the beginning of that function. This lets you distinguish among like-named static functions, as follows:

```
(fdb) list effects.mxml:myFunction
```

You can resolve ambiguous matches by extending the value of the function name or filename, as the following examples show:

Filenames:

```
(fdb) list UIOb
Ambiguous matching file names:
UIComponent.as#66
UIComponentDescriptor.as#67
UIComponentExtensions.as#68
(fdb) list UIComponent.
```

Function names:

```
(fdb) list init
Ambiguous matching function names:
init
initFromClipParameters
(fdb) list init(
```

Viewing and changing the current file

The `list` command acts on the current file by default. To change to a different file, use the `cf` command. The `cf` command has the following syntax:

```
(fdb) cf [file_name|file_number]
```

For example, to change the file to MyApp.mxml, use the following command:

```
(fdb) cf MyApp.mxml
```

If you do not specify a filename, the `cf` command lists the name and file number of the current file.

To view a list of all files used by the current application, use the `infofiles` command. For more information, see “Getting status” on page 2220.

Viewing the current working directory

Use the `pwd` command to view the file system’s current working directory, as the following example shows. This is the directory from which fdb was run.

```
(fdb) pwd
c:/Flex2SDK/bin/
```

Locating source files

Usually, fdb can find the source files for your application to display them with the `list` command. In some situations, however, you need to add a directory to the search path so that fdb can find the source files. This can be necessary, for example, when the application was compiled on a different computer than you are using to debug the application.

You use the `directory` command to add a directory to the search path. This command adds the specified directory or directories to the beginning of the list of directories that fdb searches for source files. The syntax for the `directory` command is as follows:

```
(fdb) directory path
```

For example:

```
(fdb) directory C:\MySource;C:\MyOtherSource
```

On Windows, use the semicolon character as a separator. On Macintosh and UNIX, use the colon character as a separator.

To see the current list of directories in the search path, use the `show directories` command.

Using truncated file and function names

The fdb debugger supports truncated file and function names. You can specify *`file_name`* and *`function_name`* arguments with partial names, as long as the names are unambiguous.

If you use truncated file and function names, fdb tries to map the argument to an unambiguous function name first, and then a filename. For example, `listfoo` first tries to find a function unambiguously starting with *foo* in the current file. If this fails, it tries to find a file unambiguously starting with *foo*.

Printing stack traces

Use the `bt` command to display a back trace of all stack frames. The `bt` command has the following syntax:

```
(fdb) bt
```

Getting status

Use the `info` command to get general information about the application. The `info` command has the following syntax:

```
info [options] [args]
```

The `info` command displays general information about the application being debugged. The following table describes the options of the `info` command:

Option	Description
`arguments`	Displays the argument variables of the current stack frame.
`breakpoints`	Displays the status of user-settable breakpoints.
`display`	Displays the list of autodisplay expressions.

Option	**Description**

`files [arg]`	Displays the names of all files used by the target application. This includes authored files and system files, plus generated files. Also indicates the file number for each file.
	You can use wildcards and literals to select and sort the output. The `infofiles` command supports the following:
	`info files character` Alphabetically lists files with names that start with the specified character. The following example lists all files starting with the letter V:
	`info files V`
	`info files *.extension` Alphabetically lists all files with the given extension. The following example lists all files with the as extension:
	`info files *.as`
	`info files *string*` Alphabetically lists all files with names that include string.

`functions [arg]`	Displays all function names used in this application. The `info functions` command optionally takes an argument; for example: `infofunctions` Lists all functions in all files. `infofunctions` Lists all functions in the current file. `infofunctions MyApp.mxml` Lists all functions in the MyApp.mxml file.
`handle`	Displays settings for fault handling in the debugger.
`locals`	Displays the local variables of the current stack frame.
`sources`	Displays authored source files used by the target application.
`stack`	Displays the backtrace of the stack.
`swfs`	Displays all current SWF files.
`targets`	Displays the HTTP or file URL of the target application.
`variables`	Displays all global and static variable names.

For additional information about these options, use the `help` command, as the following example shows:

```
(fdb) help info targets
```

Handling faults and catching exceptions

Use the `handle` command to specify how fdb reacts to Flash Player exceptions during execution. To view the current settings, use the `info` command, as the following example shows:

```
(fdb) info handle
```

The `handle` command has the following syntax:

```
(fdb) handle exception [action]
```

The *fault_type* is the category of fault that fdb handles. The *action* is what fdb does in response to that fault. The possible actions are `print`, `noprint`, `stop`, and `nostop`. The following table describes these actions:

Action	**Description**
`print`	Prints a message if this type of fault occurs.

Action	Description
`noprint`	Does not print a message if this type of fault occurs.
`stop`	Stops execution of the debugger if this type of fault occurs.
`nostop`	Does not stop execution of the debugger if this type of fault occurs.

You can also use the `catch` command to halt the debugger when an exception of a specified type is encountered. The debugger halts even if there is a `catch` statement that catches the exception. The syntax of the `catch` command is as follows:

```
(fdb) catch exception_type
```

The *exception_type* argument is the type of exception that stops the debugger. For example, TypeError:

```
(fdb) catch TypeError
```

To halt the debugger on any exception, you can use a wildcard (“*”), as the following example shows:

```
(fdb) catch *
```

Getting help

Use the `help` command to get information on particular topics. The `help` command has the following syntax:

```
help [topic]
```

The `help` command provides a relatively terse description of each command and its usage. The following example opens the `help` command:

```
(fdb) help
```

Type **help** followed by the command name to get the full help information, as the following example shows:

```
(fdb) help delete
```

Terminating the session

You use the `kill` and `exit` commands to end the current debugging session and exit from the fdb application. The `kill` and `exit` commands do not take any arguments. If fdb opened the default browser, you can also terminate the fdb session by closing the browser window.

To stop the current session, use the `kill` command, as the following example shows:

```
(fdb) kill
```

Using the `kill` command does not quit the fdb application. You can immediately start another session. To exit from fdb, use the `exit` command, as follows:

```
(fdb) exit
```

Logging

You can log messages at several different points in an Adobe® Flex® application’s life cycle. You can log messages when you compile the application, when you deploy it to a web application server, or when a client runs it. You can log messages on the server or on the client. These messages are useful for informational, diagnostic, and debugging activities.

About logging

When you encounter a problem with your application, whether during compilation or at run time, the first step is to gather diagnostic information to locate the cause of the problem. The source of a problem typically is in one of two places: the server web application, or the client application.

***Note:** For detailed information on debugging Adobe® AIR™ applications, see Debugging Tools in Flash Builder.*

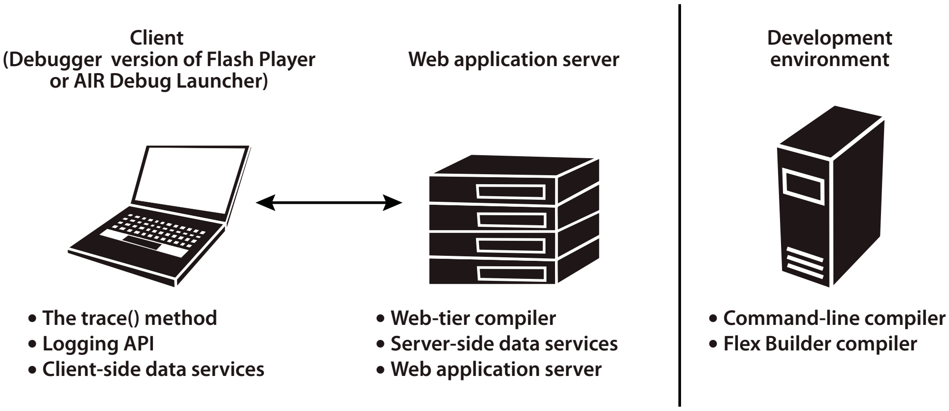
Flex includes several different logging and error reporting mechanisms that you can use to track down failures:

Client-side logging and debugging With the debugger version of Adobe® Flash® Player, or with an AIR application that you debug using ADL, you can use the global trace() method to write out messages or configure a TraceTarget to customize log levels of applications for data services-based applications. For more information, see "Client-side logging and debugging" on page 2227.

Compiler logging When compiling your Flex applications from the command line and in Adobe® Flash® Builder®, you can view deprecation and warning messages, and sources of fatal errors. For more information, see "Compiler logging" on page 2237.

Policy file logging You can log messages for all policy file interactions between a running SWF and Flash Player. For more information, see "Policy file logging" on page 2237.

The following example shows the types of logging you can do in the appropriate environment:



To use client-side debugging utilities such as the `trace()` global method and client-side data services logging, you must install and configure the debugger version of Flash Player. This is described in "Using the debugger version of Flash Player" on page 2223. The debugger version of Flash Player is not required to log compiler messages.

Using the debugger version of Flash Player

The debugger version of Flash Player is a tool for development and testing. Like the standard version of Adobe Flash Player, it runs SWF files in a browser or on the desktop in a stand-alone player. Unlike Flash Player, the debugger version of Flash Player enables you to do the following:

- Output statements and application errors to the debugger version of the Flash Player local log file by using the trace() method.
- Write data services log messages to the local log file of the debugger version of Flash Player.
- View run-time errors (RTEs).
- Use the fdb command-line debugger.
- Use the Flash Builder debugging tool.

- Use the Flash Builder profiling tool.

***Note:** Any client running the debugger version of Flash Player can view your application's `trace()` statements and other log messages unless you disable them. For more information, see "Suppressing debug output" on page 134.*

The debugger version of Flash Player lets you take advantage of the client-side logging utilities such as the `trace()` method and the logging API. You are not required to run the debugger version of Flash Player to log compiler messages because compiling does not require a player.

***Note:** ADL logs `trace()` output from AIR applications, based on the setting in mm.cfg (the same setting used by the debug version of Flash Player).*

In nearly all respects, the debugger version of Flash Player appears to be the same as the standard version of Flash Player. To determine whether or not you are running the debugger version of Flash Player, use the instructions in "Determining Flash Player version in Flex" on page 2227.

The debugger version of Flash Player comes in ActiveX, Plug-in, and stand-alone versions for Microsoft Internet Explorer, Netscape-based browsers, and desktop applications, respectively. You can find the debugger version of Flash Player installers in the following locations:

- Flash Builder: *install_dir*/Player/*os_version*
- Flex SDK: *install_dir*/runtimes/player/*os_version*/

Uninstall your current Flash Player before you install the debugger version of Flash Player. For information on installing the debugger version of Flash Player, see the Flex installation instructions.

You can enable or disable trace logging and perform other configuration tasks for the debugger version of Flash Player. For more information, see "Editing the mm.cfg file" on page 2224.

Editing the mm.cfg file

You use the settings in the mm.cfg text file to configure the debugger version of Flash Player. These settings also affect logging of `trace()` output in AIR applications running in the ADL debugger. If this file does not exist, you can create it when you first configure the debugger version of Flash Player. The location of this file depends on your operating system.

The following table shows where to create the mm.cfg file for Flash Player 10.1 and later on several operating systems:

Operating system	**Create file in ...**
Macintosh OS X	Flash Player first checks the user's home directory (~). If none is found, then Flash Player looks in /Library/Application Support/Macromedia
Windows 2000/XP	`%HOMEDRIVE%\%HOMEPATH%` The default value is "c:\Documents and settings\username". Your system administrator might map the home directory to a shared network drive. In that case, check with your system administrator to determine how to configure your debugger Player.
Windows Vista Windows 7	`%HOMEDRIVE%\%HOMEPATH%` The default value is "c:\Users\username". Your system administrator might map the home directory to a shared network drive. In that case, check with your system administrator to determine how to configure your debugger Player.
Linux	`/home/`*`username`*

On Microsoft Windows 2000, the default location of the mm.cfg file for earlier versions of Flash Player was \.

The following table lists the properties that you can set in the mm.cfg file:

Property	**Description**
`ErrorReportingEnable`	Enables the logging of error messages. Set the `ErrorReportingEnable` property to 1 to enable the debugger version of Flash Player to write error messages to the log file. To disable logging of error messages, set the `ErrorReportingEnable` property to 0. The default value is 0.
`MaxWarnings`	Sets the number of warnings to log before stopping. The default value of the `MaxWarnings` property is 100. After 100 messages, the debugger version of Flash Player writes a message to the file stating that further error messages will be suppressed. Set the `MaxWarnings` property to override the default message limit. For example, you can set it to 500 to capture 500 error messages. Set the `MaxWarnings` property to 0 to remove the limit so that all error messages are recorded.
`PolicyFileLog`	Enables the logging of policy file messages. Set `PolicyFileLog` to 1 to log policy file messages. The default value is 0. For more information on using policy file logs, see “Log file location” on page 2226.

Property	Description
`PolicyFileLogAppend`	Lets you save the contents of the policy file log file. Set the `PolicyFileLogAppend` property to 1 to save previous policy file log entries. The default value is 0. If `PolicyFileLogAppend` is not enabled, each new root-level SWF clears the log file. If PolicyFileLogAppend is enabled, the previous contents of the log file will always be kept, and the log file will grow at the end. If many different root-level SWF files are loaded during your testing, you will probably want to enable `PolicyFileLogAppend`. However, if you enable `PolicyFileLogAppend`, you will probably need to manually rename or delete the log file from time to time, since otherwise it will grow to a large size, and it will be difficult to determine where previous output ends and new output begins. For more information on using policy file logs, see "Policy file logging" on page 2237.
`TraceOutputFileEnable`	Enables trace logging. Set `TraceOutputFileEnable` to 1 to enable the debugger version of Flash Player to write trace messages to the log file. Disable trace logging by setting the `TraceOutputFileEnable` property to 0. The default value is 0.
`TraceOutputFileName`	**Note**: Beginning with the Flash Player 9 Update, Flash Player ignores the `TraceOutputFileName` property and stores the flashlog.txt file in a hard-coded location based on operating system. For more information, see "Log file location" on page 2226. Sets the location of the log file. By default, the debugger version of Flash Player writes error messages to a file named flashlog.txt, located in the same directory in which the mm.cfg file is located. Set `TraceOutputFileName` to override the default name and location of the log file by specifying a new location and name in the following form: On Macintosh OS X, you should use colons to separate directories in the TraceOutputFileName path rather than slashes. `TraceOutputFileName=<fully qualified path/filename>`

The following sample mm.cfg file enables error reporting and trace logging:

```
ErrorReportingEnable=1
```

Log file location

Beginning with the Flash Player 9 Update, you cannot modify the log file location or name. The filename is flashlog.txt, and its location is hard-coded, depending on your operating system. The following table shows the flashlog.txt file location:

Operating System	Log file location
Windows 95/98/ME/2000/XP	`C:\Documents and Settings\`*`username`*`\Application Data\Macromedia\Flash Player\Logs`
Windows Vista/Windows 7	`C:\Users\`*`username`*`\AppData\Roaming\Macromedia\Flash Player\Logs`
Macintosh OS X	`/Users/`*`username`*`/Library/Preferences/Macromedia/Flash Player/Logs/`
Linux	`/home/`*`username`*`/.macromedia/Flash_Player/Logs/`

The default log file location changed between the initial Flash Player 9 release and the Flash Player 9 Update. In the initial Flash Player 9 release, the default location was the same directory as the mm.cfg file and you could update the log file location and name through the `TraceOutputFileName` property.

Determining Flash Player version in Flex

To determine which version of Flash Player you are currently using—the standard version or the debugger version—you can use the Capabilities class. This class contains information about Flash Player and the system that it is currently operating on. To determine if you are using the debugger version of Flash Player, you can use the `isDebugger` property of that class. This property returns a Boolean value: the value is `true` if the current player is the debugger version of Flash Player and `false` if it is not.

The following example uses the `playerType`, `version`, and `isDebugger` properties of the Capabilities class to display information about the Player:

```
<?xml version="1.0"?>
<!-- logging/CheckDebugger.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        import flash.system.Capabilities;
        private function reportVersion():String {
            if (Capabilities.isDebugger) {
                return "Debugger version of Flash Player";
            } else {
                return "Flash Player";
            }
        }
        private function reportType():String {
            return Capabilities.playerType + " (" + Capabilities.version + ")";
        }
        ]]>
    </fx:Script>

    <s:Label text="{reportVersion()}"/>
    <s:Label text="{reportType()}"/>
</s:Application>
```

Other properties of the Capabilities class include `hasPrinting`, `os`, and `language`.

Client-side logging and debugging

Often, you use the `trace()` method when you debug applications to write a checkpoint message on the client, which signals that your application reached a specific line of code, or to output the value of a variable.

The debugger version of Flash Player has two primary methods of writing messages that use `trace()`:

- The global `trace()` method. The global trace() method prints Strings to a specified output log file. For more information, see “Using the global trace() method” on page 2228.

- Logging API. The logging API provides a layer of functionality on top of the `trace()` method that you can use with your custom classes or with the data service APIs. For more information, see “Using the logging API” on page 2229.

Configuring the debugger version of Flash Player to record trace() output

To record messages on the client, you must use the debugger version of Flash Player. You must also set `TraceOutputFileEnable` to 1 in your mm.cfg file. For more information on editing the mm.cfg file, see “Editing the mm.cfg file” on page 2224.

The debugger version of Flash Player sends output from the trace() method to the flashlog.txt file. The location of this file is determined by the operating system. For more information, see “Log file location” on page 2226.

You can suppress trace output by setting the `omit-trace-statements` compiler argument to `true`.

Using the global trace() method

You can use the debugger version of Flash Player to capture output from the global `trace()` method and write that output to the client log file. You can use trace() statements in any ActionScript or MXML file in your application. Because it is a global function, you are not required to import any ActionScript classes packages to use the `trace()` method.

Only debug SWF files will write out `trace()` statements. To create a debug SWF file, set the `debug` compiler argument to true.

The following example defines a function that logs the various stages of the Button control’s startup life cycle:

```
<?xml version="1.0"?>
<!-- logging/ButtonLifeCycle.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
        private function traceEvent(event:Event):void {
            trace(event.currentTarget + ":" + event.type);
        }
        ]]>
    </fx:Script>
    <s:Button id="b1" label="Click Me"
        preinitialize="traceEvent(event)"
        initialize="traceEvent(event)"
        creationComplete="traceEvent(event)"
        updateComplete="traceEvent(event)"
    />
</s:Application>
```

The following example shows the output of this simple application:

```
TraceLifecycle_3.b1:Button:preinitialize
TraceLifecycle_3.b1:Button:initialize
TraceLifecycle_3.b1:Button:creationComplete
TraceLifecycle_3.b1:Button:updateComplete
TraceLifecycle_3.b1:Button:updateComplete
TraceLifecycle_3.b1:Button:updateComplete
```

Messages that you log by using the `trace()` method should be Strings. If the output is not a String, use the `String(...)` conversion function, or use the object's `toString()` method, if one is available, before you call the `trace()` method.

To enable tracing, you must configure the debugger version of Flash Player as described in "Configuring the debugger version of Flash Player to record trace() output" on page 2228.

Using the logging API

The logging API lets an application capture and write messages to a target's configured output. Typically the output is equivalent to the global `trace()` method, but it can be anything that an active target supports.

The logging API consists of the following parts:

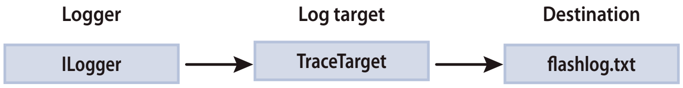
Logger The logger provides an interface for sending a message to an active target. Loggers implement the ILogger interface and call methods on the Log class. The two classes of information used to filter a message are category and level. Each logger operates under a category. A *category* is a string used to filter all messages sent from that logger. For example, a logger can be acquired with the category "orange". Any message sent using the "orange" logger only reaches those targets that are listening for the "orange" category. In contrast to the category that is applied to all messages sent with a logger, the *level* provides additional filtering on a per-message basis. For example, to indicate that an error occurred within the "orange" subsystem, you can use the error level when logging the message. The supported levels are defined by the LogEventLevel class. The Flex framework classes that use the logging API set the category to the fully qualified class name as a convention.

Log target The log target defines where log messages are written. Flex predefines a single log target: TraceTarget, which is the most commonly used log target. This log target connects the logging API to the trace system so that log messages are sent to the same location as the output of the trace() method. For more information on the `trace()` method, see "Using the global trace() method" on page 2228.

You can also write your own custom log target. For more information, see "Implementing a custom logger with the logging API" on page 2233.

Destination The destination is where the log message is written. Typically, this is a file, but it can also be a console or something else, such as an in-memory object. The default destination for TraceTarget is the flashlog.txt file. You configure this destination on the client.

The following example shows a sample relationship between a logger, a log target, and a destination:



You can also use the logging API to send messages from custom code you write. You can do this when you create a set of custom APIs or components or when you extend the Flex framework classes and you want users to be able to customize their logging. For more information, see "Implementing a custom logger with the logging API" on page 2233.

The following packages within the Flex framework are the only ones that use the logging API:

- mx.rpc.*
- mx.messaging.*
- mx.data.*

To configure client-side logging in MXML or ActionScript, create a TraceTarget object to log messages. The TraceTarget object logs messages to the same location as the output of the `trace()` statements. You can also use the TraceTarget to specify which classes to log messages for, and what level of messages to log.

The levels of logging messages are defined as constants of the LogEventLevel class. The following table lists the log level constants and their numeric equivalents, and describes each message level:

Logging level constant (int)	Description
`ALL (0)`	Designates that messages of all logging levels should be logged.
`DEBUG (2)`	Logs internal Flex activities. This is most useful when debugging an application. Select the `DEBUG` logging level to include `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL` messages in your log files.
`INFO (4)`	Logs general information. Select the `INFO` logging level to include `INFO`, `WARN`, `ERROR`, and `FATAL` messages in your log files.
`WARN (6)`	Logs a message when the application encounters a problem. These problems do not cause the application to stop running, but could lead to further errors. Select the `WARN` logging level to include `WARN`, `ERROR`, and `FATAL` messages in your log files.
`ERROR (8)`	Logs a message when a critical service is not available or a situation has occurred that restricts the use of the application. Select the `ERROR` logging level to include `ERROR` and `FATAL` messages in your log files.
`FATAL (1000)`	Logs a message when an event occurs that results in the failure of the application. Select the `FATAL` logging level to include only `FATAL` messages in your log files.

The log level lets you restrict the amount of messages sent to any running targets. Whatever log level you specify, all “lower” levels of messages are written to the log. For example, if you set the log level to `DEBUG`, all log levels are included. If you set the log level to `WARNING`, only `WARNING`, `ERROR`, and `FATAL` messages are logged. If you set the log level to the lowest level of message, `FATAL`, only `FATAL` messages are logged.

Using the logging API with data services

The data services classes are designed to use the logging API to log client-side and server-side messages.

Enable the logging API with data services

1 Create a TraceTarget logging target and set the value of one or more filter properties to include the classes whose messages you want to log. You can filter the log messages to a specific class or package. You can use wildcards (*) when defining a filter.

2 Set the log level by using the `level` property of the log target. You can also add detail to the log file output, such as the date and time that the event occurred, by using properties of the log target.

3 When you create a target within ActionScript, call the Log class’s `addTarget()` method to add the new target to the logging system. Calling the `addTarget()` method is not required when you create a target in MXML. As long as the client is using the debugger version of Flash Player and meets the requirements described in “Configuring the debugger version of Flash Player to record trace() output” on page 2228, the messages are logged.

The following example configures a TraceTarget logging target in ActionScript:

```
<?xml version="1.0"?>
<!-- logging/ActionScriptTraceTarget.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initLogging();"
    height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.logging.targets.*;
        import mx.logging.*;
        [Bindable]
        public var myData:ArrayCollection;
        private function initLogging():void {
            /* Create a target. */
            var logTarget:TraceTarget = new TraceTarget();
            /* Log only messages for the classes in the mx.rpc.* and
               mx.messaging packages. */
            logTarget.filters=["mx.rpc.*","mx.messaging.*"];
            /* Log all log levels. */
            logTarget.level = LogEventLevel.ALL;
            /* Add date, time, category, and log level to the output. */
            logTarget.includeDate = true;
            logTarget.includeTime = true;
            logTarget.includeCategory = true;
            logTarget.includeLevel = true;
            /* Begin logging. */
            Log.addTarget(logTarget);
        }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- HTTPService is in the mx.rpc.http.* package -->
        <mx:HTTPService id="srv"
            url="../assets/trace_example_data.xml"
            useProxy="false"
            result="myData=ArrayCollection(srv.lastResult.data.result)"/>
    </fx:Declarations>
    <mx:LineChart id="chart" dataProvider="{myData}" showDataTips="true">
        <mx:horizontalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
            <mx:LineSeries yField="apple" name="Apple"/>
            <mx:LineSeries yField="orange" name="Orange"/>
            <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
    </mx:LineChart>
    <s:Button id="b1" click="srv.send();" label="Load Data"/>
</s:Application>
```

In the preceding example, the `filters` property is set to log messages for all classes in the mx.rpc and mx.messaging packages. In this case, it logs messages for the HTTPService class, which is in the mx.rpc.http.* package.

You can also configure a log target in MXML. When you do this, though, you must be sure to use an appropriate number (such as 2) rather than a constant (such as `DEBUG`) for the log level.

The following example sets the values of the filters for a TraceTarget logging target by using MXML syntax:

```
<?xml version="1.0"?>
<!-- charts/MXMLTraceTarget.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="initApp();"
    height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- HTTPService is in the mx.rpc.http.* package -->
        <mx:HTTPService id="srv"
            url="../assets/trace_example_data.xml"
            useProxy="false"
            result="myData=ArrayCollection(srv.lastResult.data.result)"/>
        <mx:TraceTarget id="logTarget"
            includeDate="true"
            includeTime="true"
            includeCategory="true"
            includeLevel="true">
            <mx:filters>
                <fx:Array>
                    <fx:String>mx.rpc.*</fx:String>
                    <fx:String>mx.messaging.*</fx:String>
                </fx:Array>
            </mx:filters>
            <!-- 0 is represents the LogEventLevel.ALL constant. -->
            <mx:level>0</mx:level>
        </mx:TraceTarget>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.logging.Log;
        [Bindable]
        public var myData:ArrayCollection;

        private function initApp():void {
            Log.addTarget(logTarget);
        }
        ]]>
    </fx:Script>
    <mx:LineChart id="chart" dataProvider="{myData}" showDataTips="true">
        <mx:horizontalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
            <mx:LineSeries yField="apple" name="Apple"/>
            <mx:LineSeries yField="orange" name="Orange"/>
            <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
    </mx:LineChart>
    <s:Button id="b1" click="srv.send();" label="Load Data"/>
</s:Application>
```

Implementing a custom logger with the logging API

If you write custom components or an ActionScript API, you can use the logging API to access the trace system in the debugger version of Flash Player. You do this by defining your log target as a TraceTarget, and then calling methods on your logger when you log messages.

The following example extends a Button control. It writes log messages for the startup life cycle events, such as `initialize` and `creationComplete`, and the common UI events, such as `click` and `mouseOver`.

```
// logging/MyCustomLogger.as
package { // The empty package.
    import mx.controls.Button;
    import flash.events.*;
    import mx.logging.*;
    import mx.logging.targets.*;

    public class MyCustomLogger extends Button {
        private var myLogger:ILogger;
        public function MyCustomLogger() {
            super();
            initListeners();
            initLogger();
        }
        private function initListeners():void {
            // Add event listeners life cycle events.
            addEventListener("preinitialize", logLifeCycleEvent);
            addEventListener("initialize", logLifeCycleEvent);
            addEventListener("creationComplete", logLifeCycleEvent);
            addEventListener("updateComplete", logLifeCycleEvent);

            // Add event listeners for other common events.
            addEventListener("click", logUIEvent);
            addEventListener("mouseUp", logUIEvent);
            addEventListener("mouseDown", logUIEvent);
            addEventListener("mouseOver", logUIEvent);
            addEventListener("mouseOut", logUIEvent);
        }
        private function initLogger():void {
            myLogger = Log.getLogger("MyCustomClass");
        }
        private function logLifeCycleEvent(e:Event):void {
            if (Log.isInfo()) {
                myLogger.info(" STARTUP: " + e.target + ":" + e.type);
            }
        }
        private function logUIEvent(e:MouseEvent):void {
            if (Log.isDebug()) {
                myLogger.debug(" EVENT:   " + e.target + ":" + e.type);
            }
        }
    }
}
```

Within the application that uses the MyCustomLogger class, define a TraceTarget, as the following example shows:

```
<?xml version="1.0"?>
<!-- logging/LoadCustomLogger.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns="*" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <mx:TraceTarget level="0"
            includeDate="true"
            includeTime="true"
            includeCategory="true"
            includeLevel="true">
            <mx:filters>
                <fx:Array>
                    <fx:String>*</fx:String>
                </fx:Array>
            </mx:filters>
        </mx:TraceTarget>
    </fx:Declarations>
    <MyCustomLogger label="Click Me"/>
</s:Application>
```

After running this application, the flashlog.txt file looks similar to the following:

```
3/9/2009 18:58:05.042 [INFO] MyCustomLogger STARTUP: Main_3.mcc:MyCustomLogger:preinitialize
3/9/2009 18:58:05.487 [INFO] MyCustomLogger STARTUP:
Main_3.mcc:MyCustomLogger:initialize
3/9/2009 18:58:05.557 [INFO] MyCustomLogger STARTUP:
Main_3.mcc:MyCustomLogger:creationComplete
3/9/2009 18:58:05.567 [INFO] MyCustomLogger STARTUP: Main_3.mcc:MyCustomLogger:updateComplete
3/9/2009 18:58:05.577 [INFO] MyCustomLogger STARTUP: Main_3.mcc:MyCustomLogger:updateComplete
3/9/2009 18:58:05.577 [INFO] MyCustomLogger STARTUP: Main_3.mcc:MyCustomLogger:updateComplete
3/9/2009 18:58:06.849 [DEBUG] MyCustomLogger EVENT: Main_3.mcc:MyCustomLogger:mouseOver
3/9/2009 18:58:07.109 [DEBUG] MyCustomLogger EVENT: Main_3.mcc:MyCustomLogger:mouseDown
3/9/2009 18:58:07.340 [DEBUG] MyCustomLogger EVENT: Main_3.mcc:MyCustomLogger:mouseUp
3/9/2009 18:58:07.360 [DEBUG] MyCustomLogger EVENT: Main_3.mcc:MyCustomLogger:click
3/9/2009 18:58:07.610 [DEBUG] MyCustomLogger EVENT: Main_3.mcc:MyCustomLogger:mouseOut
```

To log a message, you call the appropriate method of the ILogger interface. The ILogger interface defines a method for each log level: `debug()`, `info()`, `warn()`, `error()`, and `fatal()`. The logger logs messages from these calls if their levels are at or under the log target's logging level. If the target's logging level is set to `all`, the logger records messages when any of these methods are called.

To improve performance, a static method corresponding to each level exists on the Log class, which indicates if any targets are listening for a specific level. Before you log a message, you can use one of these methods in an `if` statement to avoid running the code. The previous example uses the `Log.isDebug()` and `Log.isInfo()` static methods to ensure that the messages are of level `INFO` or `DEBUG` before logging them.

The previous example logs messages dispatched from any category because the TraceTarget's `filters` property is set to the wildcard character (*). The framework code sets the category of the logger to the fully qualified class name of the class in which logging is being performed. This is by convention only; any String specified when calling `Log.getLogger(x)` is the category required in a `filters` property to receive the message.

When you set the `filters` property for logging within the Flex framework, you can restrict this to a certain package or packages, or to other classes. To restrict the logging to your custom class only, add the category specified when the logger was acquired ("MyCustomLogger") to the `filters` Array, as the following example shows:

```
<mx:filters>
    <mx:Array>
        <mx:String>MyCustomLogger</mx:String>
    </mx:Array>
</mx:filters>
```

In ActionScript, you can set the `filters` property by using the following syntax:

```
traceTarget.filters = ["p1.*", "p2.*", "otherPackage*"];
```

The wildcard character can appear only at the end of a value in the Array.

The `Log.getLogger()` method sets the category of the logger. You pass this method a String that defines the category.

Note: *The Flex packages that use the logging API set the category to the current class name by convention, but it can be any String that falls within the filters definitions.*

The value of the category must fall within the definition of at least one of the filters for the log message to be logged. For example, if you set the `filters` property to something other than "*" and you use `Log.getLogger("MyCustomLogger")`, the filter Array must include an entry that matches MyCustomLogger, such as "MyCustomLogger" or "My*".

You can include the logger's category in your log message, if you set the logger's `includeCategory` property to `true`.

You can also use the ILogger interface's `log()` method to customize the log message, and you can specify the logging level in that method. The following example logs messages that use the log level that is passed into the method:

```
package { // The empty package.
    // logging/MyCustomLogger2.as
    import mx.controls.Button;
    import flash.events.*;
    import flash.events.MouseEvent;
    import mx.logging.*;
    import mx.logging.targets.*;

    public class MyCustomLogger2 extends Button {
        private var myLogger:ILogger;
        public function MyCustomLogger2() {
            super();
            initListeners();
            initLogger();
        }
        private function initListeners():void {
            // Add event listeners life cycle events.
            addEventListener("preinitialize", logLifeCycleEvent);
            addEventListener("initialize", logLifeCycleEvent);
            addEventListener("creationComplete", logLifeCycleEvent);
            addEventListener("updateComplete", logLifeCycleEvent);

            // Add event listeners for other common events.
            addEventListener("click", logUIEvent);
            addEventListener("mouseUp", logUIEvent);
            addEventListener("mouseDown", logUIEvent);
            addEventListener("mouseOver", logUIEvent);
            addEventListener("mouseOut", logUIEvent);
        }
        private function initLogger():void {
            myLogger = Log.getLogger("MyCustomClass");
        }
        private function logLifeCycleEvent(e:Event):void {
            if (Log.isInfo()) {
                dynamicLogger(LogEventLevel.INFO, e, "STARTUP");
            }
        }
        private function logUIEvent(e:MouseEvent):void {
            if (Log.isDebug()) {
                dynamicLogger(LogEventLevel.DEBUG, e, "EVENT");
            }
        }

        private function dynamicLogger(
                level:int,
                e:Event, prefix:String):void {
            var s:String = "__" + prefix + "__" + e.currentTarget +
                ":" + e.type;
            myLogger.log(level, s);
        }
    }
}
```

Compiler logging

Flex provides you with control over the output of warning and debug messages for the application and component compilers. When you compile, you can enable the message output to help you to locate and fix problems in your application.

For the command-line compiler, the settings that you use to control messages are defined in the flex-config.xml file or as command-line compiler options.

You have a high level of control over what compiler messages are displayed. For example, you can enable or disable messages such as binding-related warnings in the flex-config.xml file by using the `show-binding-warnings` option. The following example disables these messages in the flex-config.xml file:

```
<show-binding-warnings>false</show-binding-warnings>
```

You can also set this option on the command line.

For Flash Builder, you set error and warning options in the Compiler Properties dialog box. To open the Compiler Properties dialog box, select Project > Properties > Flex Compiler. You can enable or disable warnings in this dialog box. In addition, you can enter more specific options such as `show-binding-warnings` in the Additional Compiler Arguments field.

If you enable compiler messages, they are written to the console window (or System.out) by default.

Also in Flash Builder is a separate Eclipse Error Log file. This file stores messages from the Eclipse environment. The default location of this log file on Windows XP is c:\Documents and Settings*user_name*\workspace\.metadata\.log. For MacOS and Linux, the default location is also in the workspace directory, but files and directories that begin with a dot are hidden by default. As a result, you must make those files visible before you can view the log file.

For more information on the compiler logging settings, see “Viewing warnings and errors” on page 2203.

Policy file logging

Flash Player 9 and later support policy file logging. The policy file log shows messages for every event relating to policy files like the crossdomain.xml file. This includes attempts to retrieve them, successes and failures in processing them, and the fate of the requests that depend on them.

The policy file log file is named policyfiles.txt. It is in the same location as the trace log (flashlog.txt). For information on where to find these log files, see “Log file location” on page 2226.

To use policy file logging, you must use the debug version of Flash Player. In addition, you must add the following to your mm.cfg file:

```
PolicyFileLog=1    # Enables policy file logging
PolicyFileLogAppend=1  # Optional; do not clear log at startup
```

For more information about editing the mm.cfg file, see “Editing the mm.cfg file” on page 2224.

To test if you are logging policy file log messages, open any SWF file with your debug player. The application does not need to use a policy file, or even be running on the network. In the log file, you should see at least one message indicating that the root-level SWF was loaded.

For information about the meaning of the policy file log messages, see http://www.adobe.com/devnet/flashplayer/articles/fplayer9_security.html.

Versioning

You might encounter some versioning issues when working on Adobe® Flex® applications. When compiling, you can choose the version of the SDK to use and you can target specific versions of Adobe® Flash® Player. You can replicate behavior of previous SDK versions. Also, you can design applications that can load modules and other SWF files that were compiled with different SDKs.

Overview of versioning

When a new version of Flex is released, it usually includes changes to the APIs, compiler, and the user interface for Adobe® Flash® Builder™. As a result, if you were working on a project but then upgraded your version of the SDK, you might have to refactor some part of your code to be compatible with the new version. In addition, if the audience for your application is restricted in which version of Flash Player it can use, you might have to make certain concessions to ensure that your application runs without errors.

Use a different SDK

Flash Builder lets you choose any SDK, based on your needs. This lets you maintain projects in Flash Builder that have not been updated to be compatible with the latest version of the SDK, or lets you use new Flash Builder features without having to refactor older Flex projects. For more information, see "Using multiple SDKs" on page 2238.

Maintain older look and feel

The differences between some versions of Flex go beyond new features. The layout schemes and the styles of many components might have changed from one version to the next. If you want to use new Flex features but have your application look and feel the same as an older version of Flex, you can specify the version whose styles you want to use with the `compatibility-version` compiler option. For more information, see "Backward compatibility" on page 2239.

Target specific Player versions

With current Flex tools, you can target your application toward a specific version of Flash Player. To do this, you use the `target-player` compiler option. This is not the same as targeting an application to a specific *SWF version*. For more information, see "Targeting Flash Player versions" on page 2240.

Ensure sub-applications and applications work together

If you want to load a sub-application into a main application that was compiled with a different version of the compiler, you can use the `loadForCompatibility` property of the SWFLoader control. Setting this property ensures that each SWF file contains its own definitions of the linked classes, removing reliance on particular APIs that might have changed between versions. You can only use this property when the main application and sub-application were compiled with version 3.2 of the Flex framework or later. In addition, the main application must be compiled with a more recent or the same version of the compiler as the sub-applications. For more information, see "Developing multi-versioned applications" on page 213.

Using multiple SDKs

Flash Builder lets you change the version of the SDK that you use to compile your projects. You can select the SDK when you first create a project, or at any time you are working on a project. You can change the SDK for any type of Flex project, including library projects and ActionScript-only projects.

You can determine the current SDK's version by using the `-version` option on the command line. For example:

```
mxmlc -version
```

To change the SDK in Flash Builder:

1 With your project open in Flash Builder, select Project > Properties.

2 Select ActionScript Compiler in the left-hand pane.

3 Select the “Use a specific SDK” option.

4 Select the SDK you want to compile with from the drop-down list. If an SDK is not in the list, click “Configure Flex SDKs” and add the definition of the new SDK.

5 Click OK to save your changes.

When you change the SDK, Flash Builder rebuilds your application and flags any code that is no longer compatible with the selected SDK.

Flash Builder includes the Flex 4.6.0 and Flex 3.6.0 SDKs. You can also add any SDK you want by adding it to the list of available SDKs in Flash Builder..

Backward compatibility

To specify the version of the Flex compiler and framework that the output should be compatible with, use the `compatibility-version` compiler option. Possible values for this compiler option are defined as constants in the FlexVersion class. This option affects some behavior such as the theme, layout rules, padding and gaps, skins, and other style settings. In addition, it affects the rules for parsing properties files.

If you do not explicitly set the value of the `compatibility-version` option, the compiler defaults to the current SDK’s version.

In Flash Builder, you add the `compatibility-version` compiler option to the Additional Compiler Arguments field on the Flex Compiler properties panel. The following example sets the compatibility version to 3.0.0:

```
-compatibility-version=3.0.0
```

For the command-line compilers, you can either pass the `compatibility-version` compiler option on the command line or set the value in the configuration file. The following example sets the compatibility version to 4.0 in the flex-config.xml file:

```
<compiler>
    <mxml>
        <compatibility-version>4.0</compatibility-version>
    <mxml>
<compiler>
```

When you set the `compatibility-version` option, you must be sure that the configuration files that you use are compatible with the version you select.

You can programmatically access the version of the application that you are running by using the FlexVersion class. To get the current compatibility version in your application, use the `compatibilityVersionString` property of that class. This lets you conditionalize the logic in your application based on the compatibility version.

Using themes with compatibility-version

When you set the compatibility version, you should also be sure to use the appropriate theme file that matches that version. Themes are located in the frameworks/themes directory. For Flex 4.x, you do not have to specify a theme file. The default Spark theme file is designed for Flex 4.x compatibility. However, if you want the Flex 3 look and feel, you must specify the Halo theme or another theme that is compatible with the Flex 3 SDK.

The following command-line example uses the Halo theme and sets `compatibility-version` to 3.0.0:

```
mxmlc -compatibility-version=3.0.0 -theme=..\frameworks\themes\Halo\halo.swc MyApp.mxml
```

For more information on using themes, see “About themes” on page 1561.

The default style sheets for several versions of Flex are available in the framework.swc file. This SWC file contains the current default style sheet (defaults.css) and the Flex 3 default style sheet (defaults-3.0.0.css). The compiler uses the style sheet that is appropriate for the compatibility version you set. For example, if you set the `compatibility-version` option to 3.0, then the compiler uses the Flex 3 default style sheet.

Differences between SDK 3 and SDK 4.x

The differences between SDK 3 and SDK 4.x that result from setting the `compatibility-version` option include fonts, skins, and styles.

If you set `compatibility-version` to 3.0.0, differences include:

- The value of `embedAsCFF` defaults to `false`. This means that non-CFF fonts are embedded, rather than CFF fonts. For more information, see “Embedding fonts with MX components” on page 1592.
- The Halo theme should be used to define the look and feel of your application. This includes styles and skins.
- Bi-directional binding will not work.

For a list of differences when using the `compatibility-version` compiler option, see Flex 4: Backward Compatibility.

Targeting Flash Player versions

Use the `target-player` compiler option to specify the version of Flash Player that you want to target with the application. Features requiring a later version of Flash Player are not compiled into the application.

One reason for setting the target Player version is if you are unsure of Player usage penetration rates and want to choose a Player version that is lower than the most recent one. Another reason is if your users are locked in to a particular Player version, such as when they are on an intranet and cannot upgrade.

The `target-player` option has the following syntax:

```
-target-player=major_version.minor_version.revision
```

The *major_version* is required while *minor_version* and *revision* are optional. If you do not specify the *minor_version* or *revision*, the compiler uses zero.

For Flex 4.0, the only supported value of the `target-player` option is 10.0.0. For Flex 4.1, the default value is 10.1.0. For Flex 4.5, the default value is 10.2.0. For Flex 4.6, the default value is 11.1.

The minimum value for Flex 4.6 is 11.1. The minimum value for Flex 4.5 is 10.2.0. For Flex 4.0 and 4.1, the minimum value is 10.0.0 (although Flex 4.1 included Player 10.1).

If you do not explicitly set the value of this option, the compiler uses the default from the flex-config.xml file.

SWF version (advanced)

The `swf-version` compiler option specifies the SWF file format version of the output SWF file. Features requiring a later version of the SWF file format are not compiled into the application. This is different from the Player version in that it refers to the SWF specification versioning scheme.

For example, to compile an application to version 10 of the SWF specification, use the following:

```
-swf-version=10
```

This is an advanced option that you should only use if you have an understanding of the SWF specification that you are targeting. For information about the SWF specifications, visit the SWF Technology Center.

ASDoc

ASDoc is a command-line tool that you can use to create API language reference documentation as HTML pages from the ActionScript classes and MXML files in your Adobe® Flex™ application. The Adobe Flex team uses the ASDoc tool to generate the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

About the ASDoc tool

The ASDoc tool parses one or more ActionScript class definitions and MXML files, and generates API language reference documentation for all public and protected methods and properties. The ASDoc tool also recognizes many types of metadata, such as the `[Bindable]`, `[Event]`, `[Style]`, and `[Effect]` metadata tags.

You specify a single class, multiple classes, an entire namespace, or any combination as inputs to the ASDoc tool.

ASDoc generates its output as a directory structure of HTML files that matches the package structure of the input class files. Also, ASDoc generates an index of all public and protected methods and properties. To view the ASDoc output, open the index.html file in the top-level directory of the output.

Using ASDoc

To use ASDoc, run the `asdoc` command from the bin directory of your Flex installation. For example, from the bin directory, enter the following command to create output for the Flex Button class:

```
asdoc -source-path C:\flex\frameworks\projects\framework\src
    -doc-classes mx.controls.Button
    -main-title "Flex API Documentation"
    -window-title "Flex API Documentation"
    -output framework-asdoc
```

In this example, the source code for the Button class is in the directory C:\flex\frameworks\projects\framework\src\mx\controls. Use the `source-path` option to specify where the ASDoc tool looks for the source code, and the `doc-classes` option to specify the name of the class to process.

The ASDoc tool writes the output to C:\flex\bin\framework-asdoc directory, as defined by the `output` option.

The ASDoc tool supports many options for specifying the files to process. For example, instead of explicitly specifying the list of files to process, you can use the `doc-sources` option to specify a directory name. The following example runs the ASDoc tool on all files in the C:\flex\frameworks\projects\framework\src\mx\controls directory:

```
asdoc
    -doc-sources C:\a\flex\flex\sdk\frameworks\projects\framework\src\mx\controls
    -main-title "Flex API Documentation"
    -window-title "Flex API Documentation"
    -output framework-asdoc
```

To view the output, open the file C:\flex\bin\framework-asdoc\index.html. For more information on running the `asdoc` command, see "Using the ASDoc tool" on page 2261.

Creating ASDoc comments in ActionScript

A standard programming practice is to include comments in source code. The ASDoc tool recognizes a specific type of comment in your source code and copies that comment to the generated output.

Writing an ASDoc comment

An ASDoc comment consists of the text between the characters `/**` that mark the beginning of the ASDoc comment, and the characters `*/` that mark the end of it. The text in a comment can continue onto multiple lines.

Use the following format for an ASDoc comment:

```
/**
* Main comment text.
*
* @tag Tag text.
*/
```

As a best practice, prefix each line of an ASDoc comment with an asterisk (*) character, followed by a single white space to make the comment more readable, and to ensure correct parsing of comments. When the ASDoc tool parses a comment, the leading asterisk and white-space characters on each line are discarded; blanks and tabs preceding the initial asterisk are also discarded.

The ASDoc comment in the previous example creates a single-paragraph description in the output. To add additional comment paragraphs, enclose each subsequent paragraph in HTML paragraph tags, `<p></p>`. You must close the `<p>` tag, in accordance with XHTML standards, as the following example shows:

```
/**
* First paragraph of a multiparagraph description.
*
* <p>Second paragraph of the description.</p>
*/
```

All the classes that ship with Flex contain the ASDoc comments that appear in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. For example, view the mx.controls.Button class for examples of ASDoc comments.

Placing ASDoc comments

Place an ASDoc comment immediately before the declaration for a class, interface, constructor, method, property, or metadata tag that you want to document, as the following example shows for the `myMethod()` method:

```
/**
* This is the typical format of a simple
* multiline (single paragraph) main description
* for the myMethod() method, which is declared in
* the ActionScript code below.
* Notice the leading asterisks and single white space
* following each asterisk.
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

The ASDoc tool ignores comments placed in the body of a method and recognizes only one comment per ActionScript statement.

A common mistake is to put an `import` statement, or other code line or metadata tag, between the ASDoc comment for a class and the `class` declaration. Because an ASDoc comment is associated with the next ActionScript statement in the file after the comment, this example associates the comment with the import statement, not the `class` declaration:

```
/**
* This is the class comment for the class MyClass.
*/
import flash.display.*; // MISTAKE - Do not to put import statement here.
class MyClass {
}
```

Formatting ASDoc comments

The main body of an ASDoc comment begins immediately after the starting characters, `/**`, and continues until the tag section, as the following example shows:

```
/**
* Main comment text continues until the first tag.
*
* @tag Tag text.
*/
```

The first sentence of the main description of the ASDoc comment should contain a concise but complete description of the declared entity. The first sentence ends at the first period followed by a space, tab, or line terminator.

ASDoc uses the first sentence to populate the summary table at the top of the HTML page for the class. Each type of class element (method, property, event, effect, and style) has a separate summary table in the ASDoc output.

The tag section begins with the first ASDoc tag in the comment, defined by the first @ character that begins a line, ignoring leading asterisks, white space, and the leading separator characters, `/**`. The main description cannot continue after the tag section begins.

The text following an ASDoc tag can span multiple lines. You can have any number of tags, where some tags can be repeated, such as the `@param` and `@see` tags, while others cannot.

The following example shows an ASDoc comment that includes a main description and a tag section. Notice the use of white space and leading asterisks to make the comment more readable:

```
/**
* Typical format of a simple multiline comment.
* This text describes the myMethod() method, which is declared below.
*
* @param param1 Describe param1 here.
* @param param2 Describe param2 here.
*
* @return Describe return value here.
*
* @see someOtherMethod
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

For a complete list of the ASDoc tags, see “ASDoc tags” on page 2256.

Using the @private tag

By default, the ASDoc tool generates output for all public and protected elements in an ActionScript class, even if you omit the ASDoc comment for the element. The ASDoc tool ignores all elements defined as private.

To make ASDoc ignore a public or protected element, insert an ASDoc comment that contains the `@private` tag anywhere in the comment. The ASDoc comment can contain additional text along with the `@private` tag, which is also excluded from the output.

ASDoc generates output for all public classes in the list of input classes. You can specify to ignore an entire class by inserting an ASDoc comment that contains the `@private` tag before the class definition. The ASDoc comment can contain additional text along with the `@private` tag, which is also excluded from the output, as the following example shows:

```
/**
* This class is omitted from the output.
*
* @private
*/
public class MyClass {
}
```

Excluding an inherited element

By default, the ASDoc tool copies information and a link for all elements inherited by a subclass from a superclass. In some cases, a subclass might not support an inherited element. You can use the `[Exclude]` metadata tag to cause ASDoc to omit the inherited element from the list of inherited elements.

The `[Exclude]` metadata tag has the following syntax:

```
[Exclude(name="elementName", kind="property|method|event|style|effect")]
```

For example, to exclude documentation on the `click` event in the MyButton subclass of the Button class, insert the following `[Exclude]` metadata tag in the MyButton.as file:

```
[Exclude(name="click", kind="event")]
```

Using HTML tags

You can use selected HTML entities and HTML tags to define paragraphs, format text, create lists, and add anchors. For a list of the supported HTML tags, see "Summary of commonly used HTML elements" on page 2260.

Write the text of an ASDoc comment in XHTML-compliant HTML. That means your HTML syntax has to conform to XML syntax rules. For example, close all HTML tags, such as `<p>` and `<code>` tags, by inserting the closing `</p>` or `</code>` tag.

The following example comment contains HTML tags to format the output:

```
/**
* This is the typical format of a simple multiline comment
* for the myMethod() method.
*
* <p>This is the second paragraph of the main description
* of the <code>myMethod</code> method.
* Notice that you do not use the paragraph tag in the
* first paragraph of the description.</p>
*
* @param param1 Describe param1 here.
* @param param2 Describe param2 here.
*
* @return A value of <code>true</code> means this;
* <code>false</code> means that.
*
* @see someOtherMethod
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

Using special characters

The ASDoc tool can fail if your source files contain non-UTF-8 characters such as curly quotes. If it does fail, the error messages it displays refers to a line number in the class. That message helps you track down the location of the special character.

ASDoc passes all HTML tags and tag entities in a comment to the output. Therefore, if you want to use special characters in a comment, enter them using HTML code equivalents. For example, to use a less-than (<) or greater-than (>) symbols in a comment, use `&lt;` and `&gt;`. To use the at-sign (@) in a comment, use `&#64;`. Otherwise, these characters are interpreted as literal HTML characters in the output.

For a list of common HTML tags and their entity equivalents, see "Summary of commonly used HTML elements" on page 2260.

Because asterisks (*) are used to delimit comments, ASDoc does not support asterisks within a comment. To use an asterisk in an ASDoc comment, use the double tilde (~~).

Hiding text in ASDoc comments

The ASDoc style sheet contains a class called `hide`, which you use to hide text in an ASDoc comment by setting the class attribute to `hide`. Hidden text does not appear in the ASDoc HTML output, but does appear in the generated HTML file. Therefore, do not use it for confidential information. The following example uses the `hide` class:

```
/**
* Dispatched when the user presses the Button control.
* If the <code>autoRepeat</code> property is <code>true</code>,
* this event is dispatched repeatedly as long as the button stays down.
*
* <span class="hide">This text is hidden.</span>
* @eventType mx.events.FlexEvent.BUTTON_DOWN
*/
```

Rules for parsing ASDoc comments

The following rules summarize how ASDoc processes an ActionScript file:

- If an ASDoc comment precedes an ActionScript element, ASDoc copies the comment and code element to the output file.
- If an ActionScript element is not preceded by an ASDoc comment, ASDoc copies the code element to the output file with an empty description.
- If an ASDoc comment contains the `@private` ASDoc tag, the associated ActionScript element and the ASDoc comment are ignored.
- The comment text must precede any @ tags, otherwise the comment text is interpreted as an argument to an @ tag. The only exception is the `@private` tag, which can appear anywhere in an ASDoc comment.
- HTML tags, such as `<p></p>`, and `<ul></ul>`, in ASDoc comments are passed through to the output.
- HTML tags must use XML style conventions, which means there must be a beginning and ending tag. For example, close an `<li>` tag with a `</li>` tag.

Documenting ActionScript elements

You can add ASDoc comments to class, property, method, and metadata elements to document ActionScript classes. For more information on documenting MXML files, see "Documenting MXML files" on page 2251.

Documenting classes

The ASDoc tool automatically includes all public classes in its output. Place the ASDoc comment for a class just before the `class` declaration, as the following example shows:

```
/**
* The MyButton control is a commonly used rectangular button.
* MyButton controls look like they can be pressed.
* They can have a text label, an icon, or both on their face.
*/
public class MyButton extends UIComponent {
}
```

This comment appears at the top of the HTML page for the associated class.

To configure ASDoc to omit the class from the output, insert an `@private` tag anywhere in the ASDoc comment, as the following example shows:

```
/**
* @private
* The MyHiddenButton control is for internal use only.
*/
public class MyHiddenButton extends UIComponent {
}
```

Documenting properties

The ASDoc tool automatically includes all public and protected properties in its output. You can document properties that are defined as variables or defined as setter and getter methods.

Documenting properties defined as variables

Place the ASDoc comment for a public or protected property that is defined as a variable just before the `var` declaration, as the following example shows:

```
/**
* The default label for MyButton.
*
* @default null
*/
public var myButtonLabel:String;
```

A best practice for a property is to include the `@default` tag to specify the default value of the property. The `@default` tag has the following format:

```
@default value
```

This tag generates the following text in the output for the property:

```
The default value is value.
```

For properties that have a calculated default value, or a complex description, omit the `@default` tag and describe the default value in text.

ActionScript lets you declare multiple properties in a single statement. However, this does not allow for unique documentation for each property. Such a statement can have only one ASDoc comment, which is copied for all properties in the statement. For example, the following documentation comment does not make sense when written as a single declaration and would be better handled as two declarations:

```
/**
* The horizontal and vertical distances of point (x,y)
*/
public var x, y;// Avoid this
```

ASDoc generates the following documentation from the preceding code:

```
public var x
    The horizontal and vertical distances of point (x,y)

public var y
    The horizontal and vertical distances of point (x,y)
```

Documenting properties defined by setter and getter methods

Properties defined by setter and getter methods are handled in a special way by the ASDoc tool because these elements are used as if they were properties rather than methods. Therefore, ASDoc creates a property definition for an item that is defined by a setter or a getter method.

If you define a setter method and a getter method, insert a single ASDoc comment before the getter, and mark the setter as `@private`. Adobe recommends this practice because usually the getter comes first in the ActionScript file, as the following example shows:

```
/**
* Indicates whether or not the text field is enabled.
*/
public function get html():Boolean {};

/**
* @private
*/
public function set html(value:Boolean):void {};
```

The following rules define how ASDoc handles properties defined by setter and getter methods:

- If you precede a setter or getter method with an ASDoc comment, the comment is included in the output.
- If you define both a setter and a getter method, only a single ASDoc comment is needed – either before the setter or before the getter.
- If you define a setter method and a getter method, insert a single ASDoc comment before the getter, and mark the setter as `@private`.
- You do not have to define the setter method and getter method in any particular order, and they do not have to be consecutive in the source-code file.
- If you define just a getter method, the property is marked as read-only.
- If you define just a setter method, the property is marked as write only.
- If you define both a public setter and public getter method in a class, and you want to hide them by using the `@private` tag, they both must be marked `@private`.
- If you have only one public setter or getter method in a class, and it is marked `@private`, ASDoc applies normal `@private` rules and omits it from the output.
- A subclass always inherits its visible superclass setter and getter method definitions.

Documenting methods

The ASDoc tool automatically includes all public and protected methods in its output. Place the ASDoc comment for a public or protected method just before the `function` declaration, as the following example shows:

```
/**
* This is the typical format of a simple multiline documentation comment
* for the myMethod() method.
*
* <p>This is the second paragraph of the main description
* of the <code>myMethod</code> method.
* Notice that you do not use the paragraph tag in the
* first paragraph of the description.</p>
*
* @param param1 Describe param1 here.
* @param param2 Describe param2 here.
*
* @return A value of <code>true</code> means this;
* <code>false</code> means that.
*
* @see someOtherMethod
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

If the method takes an argument, include an `@param` tag for each argument to describe the argument. The order of the `@param` tags in the ASDoc comment must match the order of the arguments to the method. The `@param` tag has the following syntax:

```
@param paramName  description
```

Where *paramName* is the name of the argument and *description* is a description of the argument.

If the method returns a value, use the `@return` tag to describe the return value. The `@return` tag has the following syntax:

```
@return description
```

Where *description* describes the return value.

Documenting metadata

Flex uses metadata tags to define certain characteristics of a class. ASDoc recognizes some of these metadata tags to generate output. The metadata tags recognized by ASDoc include the following:

- `[Bindable]`
- `[DefaultProperty]`
- `[Deprecated]`
- `[Effect]`
- `[Event]`
- `[Exclude]`
- `[SkinPart]`
- `[SkinState]`
- `[Style]`

Some metadata tags take an ASDoc comment, such as `[Effect]` and `[Event]`. Other metadata tags, such as `[Bindable]` and `[DefaultProperty]` do not. If you put an ASDoc comment before a metadata tag that does not accept a comment, the comment is passed through to the next element in the class. If the next element in the class already has an ASDoc comment, the comment on the metadata tag is ignored.

For more information on the `[Exclude]` tag, see “Excluding an inherited element” on page 2244.

For more information on using these metadata tags in an application, see “Metadata tags in custom components” on page 2376.

Documenting bindable properties

A bindable property is any property that can be used as the source of a data binding expression. To mark a property as bindable, you insert the `[Bindable]` metadata tag before the property definition, or before the class definition to make all properties defined within the class bindable. The `[Bindable]` metadata tag does not take an ASDoc comment.

When a property is defined as bindable, ASDoc automatically adds the following line to the output for the property:

```
This property can be used as the source for data binding.
```

For more information on the `[Bindable]` metadata tag, see “Metadata tags in custom components” on page 2376.

Documenting default properties

The `[DefaultProperty]` metadata tag defines the name of the default property of the component when you use the component in an MXML file. The `[DefaultProperty]` metadata tag does not take an ASDoc comment.

When ASDoc encounters the `[DefaultProperty]` metadata tag, it automatically adds a line to the class description that specifies the default property. For example, see the List control in *ActionScript 3.0 Reference for the Adobe Flash Platform*.

For more information on the `[DefaultProperty]` metadata tag, see “Metadata tags in custom components” on page 2376.

Documenting deprecated properties

A class or class elements marked as deprecated is one which is considered obsolete, and whose use is discouraged. While the class or class element still works, its use can generate compiler warnings.

Insert the `[Deprecated]` metadata tag before a property, method, or class definition to mark that element as deprecated. The `[Deprecated]` metadata tag does not take an ASDoc comment.

The following example uses the `[Deprecated]` metadata tag to mark the dataProvider property as obsolete:

```
[Deprecated(replacement="MenuBarItem.data")]
public function set dataProvider(value:Object):void
{}
```

When ASDoc encounters the `[Deprecated]` metadata tag, it adds a line to the output marking the item as deprecated, and inserts a link to the replacement item.

The mxmlc command-line compiler supports the `show-deprecation-warnings` compiler option, which, when `true`, configures the compiler to issue deprecation warnings when your application uses deprecated elements. The default value is `true`.

For more information on the `[Deprecated]` metadata tag, see “Deprecated metadata tag” on page 2383.

Documenting skin states and skin parts

You use metadata tags to add information about skin parts and skin states to a class definition. Put the `[SkinPart]` metadata tag before a property definition in the source code file, and put the `[SkinState]` metadata tag at the top of the class definition.

The `[SkinPart]` metadata tag does not take an ASDoc comment. It is used to identify that a property corresponds to a skin part, as the following example shows:

```
[SkinPart(required="false")]
/**
*  A skin part that defines the  label of the button.
*/
public var labelElement:TextGraphicElement;
```

A property identified as a skin part appears in the ASDoc output in the Skin Part section, not in the Properties section. The ASDoc comment for the property appears in the Skin Part section.

The `[SkinState]` metadata tag takes an ASDoc comment describing the skin state, as the following example shows:

```
/**
*  Up State of the Button
*/
[SkinState("up")]
```

The ASDoc comment for the skin state appears in the Skin States section of the ASDoc output.

Documenting effects, events, and styles

You use metadata tags to add information about effects, events, and styles in a class definition. The `[Effect]`, `[Event]`, and `[Style]` metadata tags typically appear at the top of the class definition file. To document the metadata tags, insert an ASDoc comment before the metadata tag, as the following example shows:

```
/**
* Defines the name style.
*/
[Style "name"]
```

For events and effects, the metadata tag includes the name of the event class associated with the event or effect. The following example shows an event definition from the Flex mx.controls.Button class:

```
/**
* Dispatched when the user presses the Button control.
* If the <code>autoRepeat</code> property is <code>true</code>,
* this event is dispatched repeatedly as long as the button stays down.
*
* @eventType mx.events.FlexEvent.BUTTON_DOWN
*/
[Event(name="buttonDown", type="mx.events.FlexEvent")]
```

In the ASDoc comment for the `mx.events.FlexEvent.BUTTON_DOWN` constant, you insert a table that defines the values of the `bubbles`, `cancelable`, `target`, and `currentTarget` properties of the Event class, and any additional properties added by a subclass of Event. At the end of the ASDoc comment, you insert the `@eventType` tag so that ASDoc can find the comment, as the following example shows:

```
/**
* The FlexEvent.BUTTON_DOWN constant defines the value of the
* <code>type</code> property of the event object
* for a <code>buttonDown</code> event.
*
* <p>The properties of the event object have the following values:</p>
* <table class="innertable">
* <tr><th>Property</th><th>Value</th></tr>
* ...
* </table>
*
* @eventType buttonDown
*/
public static const BUTTON_DOWN:String = "buttonDown"
```

The ASDoc tool does several things for this event:

- In the output for the mx.controls.Button class, ASDoc creates a link to the event class specified by the `type` argument of the `[Event]` metadata tag.
- ASDoc copies the description of the `mx.events.FlexEvent.BUTTON_DOWN` constant to the description of the `buttonDown` event in the Button class.

  For a complete example, see the mx.controls.Button and mx.events.FlexEvent classes.

  For more information on the `[Effect]`, `[Event]`, and `[Style]` metadata tags, see “Metadata tags in custom components” on page 2376.

Documenting MXML files

An MXML file contains several types of elements, including MXML code, ActionScript code in `<fx:Script>` blocks, and metadata tags. The ASDoc tool supports all these element types so that you can generate ASDoc content for MXML files just as you can for ActionScript classes.

MXML files correspond to ActionScript classes where the superclass corresponds to the first tag in the MXML file. For an application file, that tag is the `<s:Application>` tag and therefore an MXML application file appears in the ASDoc output as a subclass of the Application class.

Documenting MXML elements

Use the following syntax to specify an ASDoc comment in an MXML file for an element defined in MXML:

```
<!--- asdoc comment -->
```

The comment must contain three dashes following the opening `<!` characters, and end with two dashes before the closing `>` character, as the following example shows:

```
<?xml version="1.0"?>
<!-- asdoc\MyVBox.mxml -->
<!---
    The class level comment for the component.
    This tag supports all ASDoc tags,
    and does not require a CDATA block.
-->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!---
        Comment for button
     -->
    <s:Button id="myButton" label="This button has a comment"/>
</mx:VBox>
```

In this example, the first comment is a standard XML comment ignored by ASDoc. The second comment precedes the root tag of the component and uses the three dashes to identify it as an ASDoc comment. An ASDoc comment on the root tag is equivalent to the ASDoc comment before an ActionScript class definition. Therefore, the comment appears at the top of the output ASDoc HTML file.

A leading dash at the beginning of each comment line, and any whitespace characters before the dash, are ignored, as the following example shows:

```
<!---
    - Comment for my class
    - which is implemented as mxml
-->
```

If you copy a comment from an ActionScript file that uses the /**, *, and **/ characters, those characters are also ignored, as the following example shows:

```
<!---
    /**
     * Comment for my class
     * which is implemented as mxml
     */
-->
<!---
    * Comment for my class
    * which is implemented as mxml
-->
```

All MXML elements in the file correspond to public properties of the component. The comment before the Button control defines the ASDoc comment for the public property named myButton of type mx.controls.Button.

You can use any ASDoc tags in these comments, including the `@see`, `@copy`, `@param`, `@return`, and other ASDoc comments.

The ASDoc command-line tool only processes elements of an MXML file that contain an `id` attribute. If the MXML element has an `id` attribute but no comment, the element appears in the ASDoc output with a blank comment. An MXML element with no `id` attribute is ignored, even if it is preceded by an ASDoc comment, as the following example shows:

```
<?xml version="1.0"?>
<!-- asdoc\MyVBoxID.mxml -->
<!---
    The class level comment for the component.
    This tag supports all ASDoc tags,
    and does not require a CDATA block.
    @see mx.container.VBox
-->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!---
        Comment for first button appears in the output.
     -->
    <s:Button id="myButton" label="This button has a comment"/>
    <s:Button id="myButton2"
        label="Has id but no comment so appears in output"/>
    <!---
        Comment for button with no id is ignored by ASDoc.
     -->
    <s:Button label="This button has no id"/>
</mx:VBox>
```

Comments before Definition, Library, and Private tags are ignored. Also comments inside a private block are ignored.

Documenting ActionScript in <fx:Script> blocks

Insert ASDoc comments for ActionScript code in the `<fx:Script>` block by using the same syntax as you use in an ActionScript file. The only requirement is that the ASDoc comments must be within a `CDATA` block, as the following example shows:

```
<?xml version="1.0"?>
<!-- asdoc\MyVBoxComplex.mxml -->
<!---
    The class level comment for the component.
    This tag supports all ASDoc tags,
    and does not require a CDATA block.
-->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!---
        Comment for language element - this comment will be ignored.
    -->
    <fx:Script>
        <![CDATA[
            import flash.events.MouseEvent;
            /**
             * For a method in an &lt;Script&gt; block,
             * same rules as in an AS file.
             *
             * @param eventObj The event object.
             */
            public function handleClickEvent(eventObj:MouseEvent):void {
                dispatchEvent(eventObj);
            }
            /**
             * For a property in an &lt;Script&gt; block,
             * same rules as in an AS file.
             */
            public var myString:String = new String();

        ]]>
    </fx:Script>
    <!---
        Comment for first button appears in the output.
     -->
    <s:Button id="myButton" label="This button has a comment"
        click="handleClickEvent(event);"/>
    <s:Button id="myButton2"
        label="Has id but no comment so appears in output"/>
    <!---
        Comment for button with no id is ignored by ASDoc.
     -->
    <s:Button label="This button has no id"/>
</mx:VBox>
```

Documenting MXML declarations

You can add ASDoc comments to `<fx:Declaration>` blocks in MXML, as the following example shows:

```
<fx:Declarations>
    <!---
        Specifies the skin for the first button on the ButtonBar.
        @default spark.skins.default.ButtonBarFirstButtonSkin
    -->
    <fx:Component id="firstButton">
        <s:ButtonBarButton skinClass="spark.skins.default.ButtonBarFirstButtonSkin"       />
    </fx:Component>
</fx:Declarations>
```

Documenting metadata tags

You can insert ASDoc comments for metadata tags in `<fx:Metadata>` blocks in an MXML file. For metadata tags, the ASDoc comments use the same syntax as you us in an ActionScript file. The only requirement is that the ASDoc comments must be within a `CDATA` block, as the following example shows:

```
<?xml version="1.0"?>
<!-- asdoc\MyVBoxMetaData.mxml -->
<!---
    The class level comment for the component.
    This tag supports all ASDoc tags,
    and does not require a CDATA block.
-->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <!---
        Comment for language element - this comment will be ignored.
    -->
    <fx:Script>
        <![CDATA[
            import flash.events.MouseEvent;
            /**
             * For a method in an &lt;Script&gt; block,
             * same rules as in an AS file.
             *
             * @param eventObj The event object.
             */
            public function handleClickEvent(eventObj:MouseEvent):void {
                dispatchEvent(eventObj);
            }

            /**
             * For a property in an &lt;Script&gt; block,
             * same rules as in an AS file.
             */
            public var myString:String = new String();

        ]]>
    </fx:Script>

    <fx:Metadata>
         <![CDATA[
        /**
         * Defines the default style of selected text.
         */
        [Style(name="textSelectedColor",type="Number",format="Color",inherit="yes")]

        /**
```

```
         * The component dispatches the darken event
         * when the darken property changes.
         *
         *  @eventType flash.events.Event
         */
        [Event(name="darken", type="flash.events.Event")]

        /**
         * Played when the component darkens.
         */
        [Effect(name="darkenEffect", event="darken")]
         ]]>
    </fx:Metadata>

    <!---
        Comment for first button appears in the output.
     -->
    <s:Button id="myButton" label="This button has a comment"
        click="handleClickEvent(event);"/>
</mx:VBox>
```

ASDoc tags

The following table lists the ASDoc tags:

ASDoc tag	Description	Example
@copy reference	Copies an ASDoc comment from the referenced location. The main description, @param, and @return content is copied; other tags are not copied. You typically use the @copy tag to copy information from a source class or interface not in the inheritance list of the destination class. If the source class or interface is in the inheritance list, use the @inheritDoc tag instead. You can add content to the ASDoc comment before the @copy tag. Specify the location by using the same syntax as you do for the @see tag. For more information, see "Using the @see tag" on page 2259.	@copy #stop @copy flash.display.MovieClip#stop()
@default *value*	Specifies the default value for a property, style, or effect. The ASDoc tool automatically creates a sentence in the following form when it encounters an @default tag: The default value is *value*.	@default 0xCCCCCC
@eventType *package.class.CONSTANT*@eventType *String*	Use the first form in a comment for an [Event] metadata tag. It specifies the constant that defines the value of the Event.type property of the event object associated with the event. The ASDoc tool copies the description of the event constant to the referencing class. Use the second form in the comment for the constant definition. It specifies the name of the event associated with the constant. If the tag is omitted, ASDoc cannot copy the constant's comment to a referencing class.	See "Documenting effects, events, and styles" on page 2250

@example *exampleText*	Defines a code example in an ASDoc comment. By preceding the code example with this tag, ASDoc applies style properties, generates a heading, and puts the code example in the correct location.
	Enclose the code in <listing version="3.0"></listing> tags.
	Whitespace formatting is preserved and the code is displayed in a gray, horizontally scrolling box.
	If the code inside the <listing> tags uses literal "<", ">", or"&" characters, convert them to the HTML character-code equivalent.
		@example The following code sets the volume level for your sound:
		<listing version="3.0">
		var mySound:Sound = new Sound();
		mySound.setVolume(VOL_HIGH);
		</listing>

@exampleText *string*	Use this tag in an ASDoc comment in an external example file that is referenced by the @includeExample tag. The ASDoc comment must precede the first line of the example, or follow the last line of the example. External example files support one comment before and one comment after example code.	/** * This text does not appear * in the output. * @exampleText But this does. */

ASDoc tag	**Description**	**Example**
`@includeExample` ***`textFile`***	Imports an example text file into the ASDoc output. ASDoc searches for the example file based on the package name of the class and the directory specified by the `-examples-path` option to the ASDoc tool. For example, you use the `examples-path` option to set the directory to c:\examples. To add an example for the mx.controls.Button class, place it in the mx\controls\directory under c:\examples, meaning the c:\examples\mx\controls directory. You can further qualify the location of the file by specifying a path to the `@includeExample` tag. For example, you specify the `@includeExample` as shown below: `@includeExample buttonExample/ButtonExample.mxml` ASDoc looks for an example in the directory c:\examples\mx\controls\buttonExample. If you insert this tag in the comment for a class, the example appears at the end of the output HTML file. If you insert it in the ASDoc comment for a class element, the example appears in the detailed description of the element.	`@includeExample ButtonExample.mxml`
`@inheritDoc`	Copies the comment from the superclass into the subclass, or from an interface implemented by the subclass. Use this tag in the comment of an overridden method or property. You cannot use this tag with comments on metadata tags. The main ASDoc comment, `@param`, and `@return` content are copied; other tags are not. You can add content to the comment before the `@inheritDoc` tag. When you include this tag, ASDoc uses the following search order: 1. Interfaces implemented by the current class, in alphabetical order of the package and class name, and all their base-interfaces. 2. Immediate superclass of current class. 3. Interfaces of immediate superclass and all their base-interfaces. 4. Repeat steps 2 and 3 until the Object class is reached. You can also use the `@copy` tag, but the `@copy` tag is for copying information from a source class or interface that is not in the inheritance chain of the subclass.	`@inheritDoc`
`@internal` *`text`*	Hides the text attached to the tag in the generated output. The hidden text can be used for internal comments.	`@internal Please do not publicize the undocumented use of the third parameter in this method.`

ASDoc tag	Description	Example
`@param` *`paramNamedescription`*	Adds a descriptive comment to a method parameter. The ***`paramName`*** argument must match a parameter definition in the method signature.	`@param fileName The name of the file to load.`
`@private`	Exclude the element from the generated output. To omit an entire class, put the `@private` tag in the ASDoc comment for the class; to omit a single class element, put the `@private` tag in the ASDoc comment for the element.	`@private`
`@return` *`description`*	Adds a Returns section to a method description with the specified text. ASDoc automatically determines the data type of the return value.	`@return The translated message.`
`@see` *`reference[displayText]`*	Adds a See Also heading with a link to a class element. For more information, see "Using the @see tag" on page 2259. Do not include HTML formatting characters in the arguments to the `@see` tag.	`@see flash.display.MovieClip`
`@since` ***`text`***	Adds a Since section to a class or element.	`@since January 12, 2009`
`@throws` *`package.class.className description`*	Documents an error that a method can throw.	`@throws SecurityError Local untrusted SWFs may not communicate with the Internet.`

Using the @see tag

The `@see` tag lets you create cross-references to elements within a class; to elements in other classes in the same package; and to other packages. You can also cross-reference URLs outside ASDoc. The `@see` tag has the following syntax:

```
@see reference [displayText]
```

where *reference* specifies the destination of the link, and *displayText* optionally specifies the link text. The location of the destination of the `@see` tag is determined by the prefix to the *reference* attribute:

- `#` ASDoc looks in the same class for the link destination.
- *ClassName* ASDoc looks in a class in the same package for the link destination.
- *PackageName* ASDoc looks in a different package for the link destination.
- `effect` ASDoc looks for an effect property for the link destination.
- `event` ASDoc looks for an event property for the link destination.
- `style` ASDoc looks for a style property for the link destination.

***Note:** You cannot insert HTML tags in reference. However, you can add an HTML link without using the `@see` tag by inserting the HTML code in the ASDoc comment.*

The following table shows several examples of the `@see` tag:

Example	Result
@see "Just a label"	Text string
@see http://www.cnn.com	External web site
@see package-detail.html	Local HTML file
@see AccessibilityProperties	Class in same package
@see flash.display.TextField	Class in different package
@see flash.ui.ContextMenu#customItems	Property in class in different package
@see mx.containers.DividedBox#style:dividerAffordance	Style property in class in different package
@see #updateProperties()	Method in same class as @see tag
@see flash.ui.ContextMenu#clone()	Method in class in different package
@see flash.util.#clearInterval()	Package method in flash.util
@see mx.controls.Buttont#style:horizontalGap	Style property in Button class.
@see mx.containers.Accordion#event:change	Event in Accordion class.
@see mx.core.UIComponent#effect:creationCompleteEffect	Effect property in UIComponent class.

Summary of commonly used HTML elements

Write the text of an ASDoc comment in XHTML-compliant HTML. That means your HTML syntax has to conform to XML syntax rules. For example, close all HTML tags, such as `<p>` and `<code>` tags, by inserting the closing `</p>` or `</code>` tag.

The following table lists commonly used HTML tags and character codes within ASDoc comments:

Tag or Code	Description

`<p>`	Starts a new paragraph. You must close `<p>` tags.
	Do not use `<p>` for the first paragraph of a doc comment (the paragraph after the opening `/**`) or the first paragraph associated with a tag. Use the `<p>` tag for subsequent; for example:
	`/**`
	`* The first sentence of a main description.`
	`*`
	`* <p>This line starts a new paragraph.</p>`
	`*`
	`* <p>This line starts a third paragraph.</p>`
	`*/`
	ASDoc ignores white space in comments. To add white space for readability in the AS file, do not use the `<p>` tag but add blank lines.

`class="hide"`	Hides text. Use this tag if you want to add documentation to the source file but do not want it to appear in the output.
`<listing>`	Indicates a program listing (sample code). Use this tag to enclose code snippets that you format as separate paragraphs, in monospace font, and in a gray background to distinguish the code from surrounding text. Close all `<listing>` tags.

Tag or Code	Description
<pre>	Formats text in monospace font, such as a description of an algorithm or a formula. Do not use <br/> tags at end of line. Use <listing> tag for code snippets.
<br/>	Adds a line break. You must close this tag. Comments for most tags are automatically formatted; you do not generally have to add line breaks. To create additional white space, add a new paragraph instead. This tag might not be supported in the future, so use it only if necessary.
<ul>, <li>	Creates a list. You must close these tags.
<table><th><tr><td>	Creates a table. For basic tables that conform to ASDoc style, set the class attribute to `innertable`. Avoid setting any special attributes. Avoid nesting structural items, such as lists, within tables. ASDoc uses a standard CSS stylesheet that has definitions for the <table>, <th>, <tr> and <td> tags. You must close these tags. Use <th> for header cells instead of <td>, so the headers get formatted correctly.
<img>	Inserts an image. To create the correct amount of space around an image, enclose the image reference in <p></p> tags. Captions are optional; if you use a caption, make it boldface. You must close the <img> tag by ending it with />, as the following example shows: `<img src = "../../images/matrix.jpg" />`
<code>	Applies monospace formatting. You must close this tag.
<strong>	Applies bold text formatting. You must close this tag.
<em>	Applies italic formatting. You must close this tag.
&lt;	Less-than operator (<) . Ensure that you include the final semicolon (;).
&gt;	Greater-than operator (>). Ensure that you include the final semicolon (;).
&#38;	Ampersand (&). Do not use &amp;. Ensure that you include the final semicolon (;).
&#42;	Do not use a literal "*" character in the body of a comment; instead, insert the HTML character code &#42;.
&#x2014;	Em dash. Ensure that you include the final semicolon (;).
&#x99;	Trademark symbol (™) that is not registered. This character is superscript by default, so do not enclose it in <sup> tags. Ensure that you include the final semicolon (;).
&#xA0;	Nonbreaking space. Ensure that you include the final semicolon (;).
&#xAE;	Registered trademark symbol (®). Enclose this character in <sup> tags to make it superscript. Ensure that you include the final semicolon (;).
&#xB0;	Degree symbol. Ensure that you include the final semicolon (;).
&#64;	Do not use an @ sign in an ASDoc comment; instead, insert the HTML character code: &#64;.

Using the ASDoc tool

The ASDoc tool, `asdoc`, is in the flex\bin directory of a Flex installation. To run the ASDoc tool, make sure that you either first change to the bin directory, or you add the bin directory to your system path.

To see a list of the command-line options available to the ASDoc tool, use the `-help list` option, as the following example shows:

```
asdoc -help list
```

By default, the ASDoc tool compiles its input files against the library SWC files in the flex\frameworks\libs directory in your Flex installation. If you must add additional SWC files to compile your code, you can add them by using the `library-path` option to specify the directory containing the SWC files:

```
asdoc ... -library-path+=C:\myLibs
```

You can also use a Flex Ant task to run the ASDoc tool. For more information, see "Using the asdoc task" on page 2353.

Defining the input files to the ASDoc tool

Use the following options to specify the list of classes processed by the `asdoc` command: `doc-sources`, `doc-classes`, and `doc-namespaces`. The `doc-classes` and `doc-namespaces` options require you to specify the `source-path` option to specify the root directory of your files.

Note: *The examples below assume that you installed Flex in the c:\flex directory on your machine. All the Flex source code is then available in the C:\flex\frameworks\projects directory.*

The most basic example is to specify the path to a single class by using the `doc-sources` option, as the following example shows:

```
asdoc -doc-sources C:\flex\frameworks\projects\framework\src\mx\controls\Button.as
    -output framework-asdoc
```

This example generates ASDoc content for the Flex Button control shipped with Flex, and writes the output to the flex\bin\framework-asdoc directory.

You can specify multiple input class files, separated by spaces, as the following example shows:

```
asdoc -doc-sources C:\flex\frameworks\projects\framework\src\mx\controls\Button.as
    C:\flex\frameworks\projects\framework\src\mx\controls\ButtonBar.as
    -output framework-asdoc
```

The two previous examples use the `output` option to the directory that contains the ASDoc output. You can specify an absolute or relative path. In the previous example, the output directory is named framework-output relative to the current working directory. To view the output of your ASDoc build, open the index.html file in the output directory.

The `doc-sources` option takes a directory as an argument, as well as a file list. If you specify a directory, the ASDoc tool generates output for all files in the specified directory and any subdirectories. Use this option to build ASDoc output for all the files in the C:\flex\frameworks\projects\framework\src\mx\controls directory, as the following example shows:

```
asdoc -doc-sources C:\flex\frameworks\projects\framework\src\mx\controls
    -output framework-asdoc
```

You can specify multiple directories, separated by spaces.

Use the `doc-classes` option to specify the package and class name of a file to process. Use the `doc-classes` option with the `source-path` option, as the following example shows:

```
asdoc -source-path C:\flex\frameworks\projects\framework\src
    -doc-classes mx.controls.Button mx.controls.ButtonBar
    -output framework-asdoc
```

The `source-path` option adds directories to the source path. The ASDoc tool searches directories in the source path for the files to process. The value to the `doc-classes` option is a space delimited list of input files that use dot notation to specify the package name of each class.

You can combine the `doc-sources` and `doc-classes` options to specify the input to the ASDoc tool, as the following example shows:

```
asdoc -source-path C:\flex\frameworks\projects\framework\src
    -doc-classes mx.controls.Button mx.controls.ButtonBar
    -doc-sources C:\flex\frameworks\projects\framework\src\mx\validators
    -output framework-asdoc
```

In this example, you compile all the validator classes as well as the Button and ButtonBar components.

Compiling dependent files

When performing a build, the ASDoc tool compiles the input files and also attempts to compile any dependent files referenced by the input files. For example, you specify class A as an input by using the `doc-classes` option. If class A imports class B, both class A and class B are compiled and included in the ASDoc output.

The following example specifies only the mx.controls.Button class as input:

```
asdoc -source-path C:\flex\frameworks\projects\framework\src
    -doc-classes mx.controls.Button
    -output framework-asdoc
```

The output of the build includes the mx.controls.Button class, plus any class referenced by the Button class, and any classes referenced by classes referenced by Button. The compiler uses the `source-path` option to locate these dependent classes.

If you set the `exclude-dependencies` option to `true`, dependent classes found when compiling the input classes are not documented. The default value is `false`, which means any classes that would normally be compiled along with the specified classes are documented.

The following example generates ASDoc content only for the Button class:

```
asdoc
    -source-path C:\flex\frameworks\projects\framework\src
    -doc-classes mx.controls.Button
    -output framework-asdoc
    -exclude-dependencies=true
```

Setting the `exclude-dependencies` option to `true` improves the performance of the ASDoc tool because you do not have to build output for all dependent classes.

***Note:** You cannot use `exclude-dependencies` with input class specified by the `doc-sources` option.*

Adding package descriptions

To add package descriptions to the output, you can use the `package` or `package-description-file` options of ASDoc.

The `package` option lets you specify the descriptions on the ASDoc command line. You can specify more than one `package` option. The following example adds two package descriptions to the output:

```
asdoc -doc-sources my_dir -output myDoc
    -package com.my.business "Contains business classes and interfaces"
    -package com.my.commands "Contains command base classes and interfaces"
```

If you have many packages, you can use the `package-description-file` option to specify an XML file that contains the descriptions, as the following example shows:

```
asdoc -source-path C:\\flex\sdk\frameworks\projects\framework\src
    -doc-classes mx.controls.Button mx.controls.ButtonBar
    -package-description-file myPackages.xml -output myDoc
```

In this example, the package descriptions are located in the myPackages.xml file. The file specified by the `package-description-file` option has the following format:

```
<overviews>
    <packages>
        <package name="package.name1">
            <shortDescription>
                <![CDATA[
                    Short description appears on the All Packages page.
                ]]>
            </shortDescription>
            <description>
                <![CDATA[
                    Long description appears on the package page.
                ]]>
            </description>
        </package>
    </packages>
</overviews>
```

For example:

```
<overviews>
    <packages>
        <package name="mx.core">
            <shortDescription>
                <![CDATA[
                    The &lt;b&gt;mx.core package&lt;/b&gt; contains the
                    base classes and interfaces.
                ]]>
            </shortDescription>
            <description>
                <![CDATA[
                    The mx.core package contains the
                    base classes and interfaces,
                    such as UIComponent, used by Flex.
                ]]>
            </description>
        </package>
        <package name="mx.controls">
            <shortDescription>
                <![CDATA[
                    The mx.controls package contains
                    the Flex user-interface controls.
                ]]>
            </shortDescription>
            <description>
                <![CDATA[
                    The mx.controls package contains
                    the Flex user-interface controls.
                ]]>
            </description>
        </package>
    </packages>
</overviews>
```

Notice that the HTML tag for bold text is entered by using the HTML character code for the “<“ and “>” characters.

Using a manifest file as input

If your source code is packaged for distribution as a SWC file, you can use a manifest file to define the content of the SWC file. You can use a manifest file as input to the ASDoc tool to specify the input file list as the following example shows:

```
asdoc
    -source-path C:\a\flex\flex\sdk\frameworks\projects\framework\src
    -doc-classes FrameworkClasses
    -namespace http://www.adobe.com/2006/mxml
C:\flex\sdk\frameworks\projects\framework\manifest.xml
    -doc-namespaces http://www.adobe.com/2006/mxml
    -output framework-asdoc
```

The preceding command line generates ASDoc content for all classes in the Flex framework.swc file. Notice that your specify the FrameworkClasses class file as input using the `doc-classes` option, and the manifest file by using the `doc-namespace` option. Most Flex SWC files are represented by a class file and a manifest file. Therefore, to build ASDoc for the SWC file, you specify both as input.

For more information on manifest files, see "About manifest files" on page 2206.

Excluding classes

All the classes specified by the `doc-classes`, `doc-sources`, and `doc-namespaces` options are documented, with the following exceptions:

- If you specified the class by using the `exclude-classes` option, the class is not documented. You must specify the package name of the files to omit, such as mx.controls.Button, separated by spaces.
- If the ASDoc comment for the class contains the `@private` tag, the class is not documented.
- If the class is found in a SWC, the class is not documented.

In the following example, you generate output for all classes in the current directory and its subdirectories, except for the two classes comps\PageWidget and comps\ScreenWidget.as:

```
asdoc -source-path . -doc-sources . -exclude-classes comps.PageWidget comps.ScreenWidget
```

The excluded classes are still compiled along with all the other input classes; only their content in the output is suppressed.

Use the `exclude-sources` option to exclude a file from being input to the compilation. This option is different from the `exclude-classes` option which you use to exclude a class from the output. However, the `exclude-classes` option still compiles the specified class. When you specify a class by using the `exclude-sources` option, the class is not even compiled.

***Note:** You can only exclude classes added by the `doc-sources` option, not classes added by the `doc-namespaces` or `doc-classes` options.*

For example, the Flex mx/core package contains several include files that are not stand-alone classes, but files included by other classes. If you specify `doc-sources` for mx/core, you get compiler errors because the compiler tries to process all the files in the directory. Instead, you can use the `exclude-source` option to specify the full path to the files to ignore from the compilation, as the following example shows. Specify multiple paths separated by spaces, as the following example shows:

```
asdoc
    -doc-sources C:\a\flex\flex\sdk\frameworks\projects\framework\src\mx\core
    -exclude-sources C:\a\flex\flex\sdk\frameworks\projects\framework\src\mx\core\Version.as
```

Excluding a namespace

To exclude an entire namespace from ASDoc, edit the ASDoc_Config_Base.xml file in the asdoc\templates directory of your Flex installation.

***Note:** Do not edit any other settings in the ASDoc_Config_Base.xml file. Set all other options from the asdoc command line.*

For each namespace that you want to exclude, add a new `<namespace>` tag to the `<namespaces>` tag in the ASDoc_Config_Base.xml file, as the following example shows:

```
<namespaces hideAll="false">
...
    <namespace hide="true">my_custom_namespace</namespace>
</namepaces>
```

Adding ASDoc XML files to a SWC files

Flex SWC files include a docs folder that contains the intermediate XML files created by the ASDoc tool. These files are used by Adobe® Flash® Builder™ to show code hints and documentation.

You can add these XML files to your own SWC files. Normally, these intermediate XML files are deleted when the ASDoc tool completes. If you run the ASDoc tool with the `keep-xml` and `skip-xsl` options set to `true`, the XML files are retained.

By default, the ASDoc tool writes these XML files to the tempdita directory in the output directory of your build. You have to manually package them in your SWC file.

Handling ASDoc errors

The ASDoc tool compiles all the input source files to generate its output. Therefore, your source code must be valid to generate ASDoc output. The ASDoc tool writes any compilation, such as syntax errors, errors to the console window.

Errors in ASDoc comments are not compilation errors but they do cancel the ASDoc build. For example, you can cause an error in an ASDoc comment by omitting a closing `</code>` or `</p>` tag. The source code still compiles, but ASDoc fails because it cannot generate the output.

The following example shows the error message for an ASDoc comment that omits a closing `</code>` tag:

```
[Fatal Error] :10:5: The element type "code" must be terminated by the matching end-tag
"</code>".
Encountered not well-formed text. Please see C:\flex\bin\framework-asdoc\validation_errors.log
for details.
```

The error message describes the condition that caused the error, and specifies to view the validation_errors.log file for more information. The validation_errors.log file contains additional information that describes the error and the location of the error in the input file.

Use the `lenient` option to configure the ASDoc tool to generate output even when it encounters an error in an ASDoc comment. When specified, the `lenient` option causes the tool to omit the incorrect ASDoc comment from the output, but to complete the build. The following example uses the `lenient` option:

```
asdoc -doc-sources C:\flex\frameworks\projects\framework\src\mx\controls\Button.as
    -lenient
    -output framework-asdoc
```

With the `lenient` option, you see the same error message, and the ASDoc tool writes the same information to the validation_errors.log file. But, the tool generates the output, as the following message shows:

```
[Fatal Error] :10:5: The element type "code" must be terminated by the matching end-tag
"</code>".
Encountered not well-formed text. Please see C:\flex\bin\framework-asdoc\validation_errors.log
for details.
Documentation was created in C:\flex\bin\framework-asdoc\
```

Using a configuration file with the ASDoc tool

Depending on the number of input files, the input specification to the ASDoc tool can get complex. To simplify it, you can use a configuration file with the ASDoc tool. The configuration file is an XML file that lets you specify the command-line arguments to the tool in a file. YOu then run the tool as show below:

```
asdoc -load-config+=configFileName.xml -output framework-asdoc
```

By default, the ASDoc tool uses the flex\frameworks\flex-config.xml configuration file. This configuration file contains many settings, including the location of the Flex SWC files required to compile the input files. You can want to look at this file for an example of the types of information that you can write to a configuration file.

The previous example uses the `load-config` option to specify the name of the configuration file. Notice that this option uses the += syntax to add configFileName.xml to the list of configuration files so that the tool still includes the flex\frameworks\flex-config.xml configuration file. If you use the = syntax to specify the name of the configuration file, you must also define much of the information in the flex\frameworks\flex-config.xml configuration file.

The following configuration file creates the same output as the example in the section "Using a manifest file as input" on page 2265 to generates ASDoc content for all classes in the Flex framework.swc file:

```
<?xml version="1.0"?>
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
    <compiler>
        <source-path>
            <path-element>C:\flex\frameworks\projects\framework\src</path-element>
        </source-path>
        <namespaces>
            <namespace>
                <uri>http://www.adobe.com/2006/mxml</uri>
                <manifest>C:\flex\frameworks\projects\framework\manifest.xml</manifest>
            </namespace>
        </namespaces>
    </compiler>
    <doc-classes>
        <class>FrameworkClasses</class>
    </doc-classes>

    <doc-namespaces>
        <uri>http://www.adobe.com/2006/mxml</uri>
    </doc-namespaces>
</flex-config>
```

For more information on configuration files, see "About configuration files" on page 2171.

Options to the asdoc tool

The `asdoc` tool works the same way as the mxmlc compiler and takes all the same command-line options. For more information on mxmlc, see "Flex compilers" on page 2164.

The following table lists the command-line options specific to the `asdoc` tool:

Option	Description
`-doc-classes` *`path-element [...]`*	A list of classes to document. These classes must be in the source path. This is the default option. This option works the same way as does the -include-classes option for the `compc` component compiler. For more information, see "Using compc, the component compiler" on page 2194.
`-doc-namespaces` *`uri manifest`*	A list of URIs to document. The classes must be in the source path. You must include a URI and the location of the manifest file that defines the contents of this namespace. This option works the same way as does the -include-namespaces option for the `compc` component compiler. For more information, see "Using compc, the component compiler" on page 2194.
`-doc-sources` *`path-element [...]`*	A list of files to document. If a directory name is in the list, it is recursively searched. This option works the same way as does the -include-sources option for the `compc` component compiler. For more information, see "Using compc, the component compiler" on page 2194.
`-examples-path` ***`path-element`***	Specifies the location of the include examples used by the `@includeExample` tag. This option specifies the root directory. The examples must be located under this directory in subdirectories that correspond to the package name of the class. For example, you specify the `examples-path` as c:\myExamples. For a class in the package myComp.myClass, the example must be in the directory c:\myExamples\myComp.myClass.
`-exclude-classes` *`string`*	A list of classes not documented. You must specify individual class names. Alternatively, if the ASDoc comment for the class contains the `@private` tag, is not documented.
`-exclude-dependencies true|false`	Whether all dependencies found by the compiler are documented. If `true`, the dependencies of the input classes are not documented. The default value is `false`.
`-exclude-sources` ***`path-element [...]`***	Exclude a file from compilation. This option is different from the `-exclude-classes` option which you use to exclude a class from the output. However, the `-exclude-classes` option still compiles the specified class. You can only exclude classes added by the `doc-sources` option, not classes added by the `doc-namespaces` or `doc-classes` options.
`-footer` *`string`*	The text that appears at the bottom of the HTML pages in the output documentation.
`-keep-xml=false|true`	When `true`, retain the intermediate XML files created by the ASDoc tool. The default value is `false`.
`-left-frameset-width` *`int`*	An integer that changes the width of the left frameset of the documentation. You can change this size to accommodate the length of your package names. The default value is 210 pixels.
`-lenient`	Ignore XHTML errors (such as a missing </p> tag) and produce the ASDoc output. All errors are written to the validation_errors.log file.
`-main-title` *`"string"`*	The text that appears at the top of the HTML pages in the output documentation. The default value is "`API Documentation`".
`-output` *`string`*	The output directory for the generated documentation. The default value is "`asdoc-output`".

Option	**Description**
`-package` *`name"description"`*	The descriptions to use when describing a package in the documentation. You can specify more than one `package` option. The following example adds two package descriptions to the output: `asdoc -doc-sources my_dir -output myDoc -package com.my.business "Contains business classes and interfaces" -package com.my.commands "Contains command base classes and interfaces"`
`-package-description-file`*fileName*	Specifies an XML file containing the package descriptions.
`-skip-xsl=false|true`	When `true`, configures the ASDoc tool to generate the intermediate XML files only, and not perform the final conversion to HTML. The default value is `false`.
`-strict=false|true`	Disable strict compilation mode. By default, classes that do not define constructors, or contain methods that do not define return values cause compiler failures. If necessary, set `strict` to `false` to override this default and continue compilation.
`-templates-path` *`string`*	The path to the ASDoc template directory. The default is the asdoc/templates directory in the ASDoc installation directory. This directory contains all the HTML, CSS, XSL, and image files used for generating the output.
`-window-title` *`"string"`*	The text that appears in the browser window in the output documentation. The default value is "`API Documentation`".

The `asdoc` command also recognizes the following options from the compc component compiler:

- `-source-path`
- `-library-path`
- `-namespace`
- `-load-config`
- `-actionscript-file-encoding`
- `-help`
- `-advanced`
- `-benchmark`
- `-strict`
- `-warnings`

  For more information, see “Using mxmlc, the application compiler” on page 2174. All other application compiler options are accepted but ignored so that you can use the same command-lines and configuration files for the ASDoc tool that you can use for mxmlc and compc.

Chapter 9: Testing and automation

Creating applications for testing

You can create applications and components that can be tested with automated testing tools such as HP QuickTest Professional™ (QTP). The information in this topic is intended for Adobe® Flex™ developers who write applications that are tested by Quality Control (QC) professionals who use these testing tools. For information on installing and running the Flex plug-in with QTP, QC professionals should see *Testing Adobe Flex Applications with HP QuickTest Professional*.

Full support for the Flex automation features is included in Adobe® Flash® Builder™ Premium. Adobe Flash Builder Standard allows only limited use of this feature.

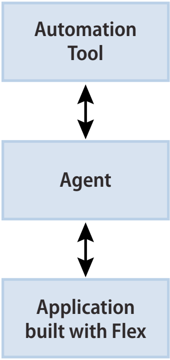
About automating applications with Flex

The automation feature provides developers with the ability to create applications that use the automation APIs. You can use these APIs to create automation agents or to ensure that your applications are ready for testing. In addition, the automation feature includes support for the QTP automation tool.

When working with the automation APIs, you should understand the following terms:

- *automation agent* (or, simply, *agent*) — An agent facilitates communication between an application and an automation tool. The Flex Automation Package includes a plugin that acts as an agent between your applications and the QTP testing tool.
- *automation tool* — Automation tools are applications that use the data that is derived through the agent. These tools include QTP, Omniture, and Segue.
- *delegates* — Flex framework components are instrumented by attaching a delegate class to each component at run time. The *delegate class* defines the methods and properties required to perform instrumentation.

The following illustration shows the relationship between an application, an agent, and an automation tool.



As this illustration shows, the automation tool uses an agent to communicate with the application built with Flex. The agent can be an ActiveX control or other type of utility that fascilitates the interaction between the tool and the application.

The Flex automation feature includes the following:

- Automation libraries — The SWC files in the libs/automation directory are the implementations of the automation API for the Flex framework components. The following table describes these SWC files.

SWC file	Description
automation.swc	Provides the delegates for the core Flex classes. This includes the Halo component set.
automation_agent.swc	Provides the classes for creating a custom agent.
automation_dmv.swc	Provides the delegates for the Flex charting and AdvancedDataGrid classes.
automation_air.swc	Provides the delegates for AIR.
automation_airspark.swc	Provides the delegates for the Spark components that are used in AIR.
automation_spark.swc	Provides the delegates for the Spark component set.
automation_flashflexkit.swc	Provides the delegates for the FlashFlexKit.
qtp.swc	Provides the classes that allow QTP to communicate with an application built with Flex.
qtp_air.swc	Provides the classes that allow QTP to communicate with AIR applications.

- QTP files — The QTP files let QTP and applications built with Flex communicate directly. You can only use these files if you also have QTP. These files include the QTP plug-in, a QTP demonstration video, a QTP-specific environment XML file, and the QTP-specific libraries, qtp.swc and qtp_air.swc. For more information, see *Testing Adobe Flex Applications with HP QuickTest Professional*.

Tasks and techniques for testable applications overview

Flex developers should review the information about tasks and techniques for creating testable applications, and then update their applications accordingly. QC testing professionals who use QTP should use the documentation provided in the separate book, *Testing Adobe Flex Applications with HP QuickTest Professional*. That document is available for download with the Flex plug-in for QTP.

Use the following general steps to create a testable application:

1. Review the guidelines for creating testable applications. For more information, see "Creating test-friendly applications" on page 2274.
2. Prepare the application to load the automation classes at run time or compile time.
   - To create an application that loads the automation classes at run time, you compile it as normal. At run time, you load your application into a wrapper SWF file that has the automation libraries built in. This wrapper SWF file uses the SWFLoader to load your application SWF file that you plan to test only at run time. For more information, see "Using run-time loading" on page 2273.
   - To compile an application that includes the automation classes, you include the needed automation libraries at compile time. Compile the application with the automation SWC files by using the compiler's `include-libraries` option. For information on the compilation process, see "Using compile-time loading" on page 2272.
3. Prepare customized components for testing. If you have custom components that extend UIComponent, make them testable. For more information, see "Instrumenting custom components" on page 2284.
4. Create an HTML wrapper that follows proper application naming practices. For more information, see "Writing the wrapper" on page 2277.

5 Deploy the application's assets to a web server. Assets can include the SWF file; HTML wrapper and related files; external assets such as theme files, graphics, and video files; module SWF files; resource modules; CSS SWF files; and run-time shared libraries (RSLs). For information about what files to deploy with your application, see "Deployment checklist" on page 2548.

Using compile-time loading

When you embed functional testing classes in your application SWF file at compile time, you increase the size of the SWF file. If the size of the application SWF file is not important, you can use the same SWF file for functional testing and deployment. If the size of the SWF file is important, you typically generate two SWF files: one with functional testing classes embedded and one without.

To compile a testable application, you must reference the necessary automation SWC files with the `include-libraries` compiler option. Typically, this includes the automation.swc and automation_spark.swc files. If your application uses charts or the AdvancedDataGrid classes, you must also add the automation_dmv.swc file. You might also be required to add automation tool-specific SWC files; for example, for QTP, you must also add the qtp.swc file to your application's library path. For a complete list of automation SWC files, see "About automating applications with Flex" on page 2270.

By default, Flash Builder includes the automation SWC files in its library path, but you must add these libraries by using the `include-libraries` compiler option.

Including automation SWC files

To include the SWC file in your application, you can add them to the compiler's configuration file or as a command-line option. For the SDK, the configuration file is located at *sdk_install_dir*/frameworks/flex-config.xml. To add the automation.swc and automation_spark.swc libraries, for example, add the following lines to the configuration file:

```
<include-libraries>
    <library>/libs/automation/automation.swc</library>
    <library>/libs/automation/automation_spark.swc</library>
</include-libraries>
```

You must uncomment the `include-libraries` code block. By default it is commented out in the configuration file.

You can also specify the location of the SWC files when you use the command-line compiler with the `include-libraries` compiler option. The following example adds the automation.swc and automation_spark.swc files to the application:

```
mxmlc -include-libraries+=../frameworks/libs/automation/automation.swc;
    ../frameworks/libs/automation/automation_spark.swc MyApp.mxml
```

Explicitly setting the `include-libraries` option on the command line overwrites, rather than appends, any existing libraries that you include in the configuration file. As a result, if you add the automation.swc and automation_spark.swc files by using the `include-libraries` option on the command line, ensure that you use the += operator. This does not overwrite the existing libraries that might be included.

To add automated testing support to a Flash Builder project, you also add the SWC files to the `include-libraries` compiler option.

Add SWC files to Flash Builder projects

1 In Flash Builder, select your Flex project in the Navigator.

2 Select Project > Properties. The Properties dialog box appears.

3 Select Flex Compiler in the tree to the left. The Flex Compiler properties panel appears.

4 In the "Additional compiler arguments" field, enter the following command:

```
-include-
libraries+="sdks\4.6.0\frameworks\libs\automation\automation.swc","sdks\4.6.0\frameworks\l
ibs\automation\automation_spark.swc"
```

In Flash Builder, the entries in the `include-libraries` compiler option are relative to the Flash Builder installation directory; the default location of this directory on Windows is C:\Program Files\Adobe\Flash Builder 4.6.

5 Click OK to save your changes.

Deploying the application

When you create the final release version of your application, you recompile the application without the references to the automation SWC files.

If you do not deploy your application to a server, but instead request it by using the file protocol or run it from within Adobe Flash Builder, you must put the SWF file into the local-trusted sandbox. This requires configuration information that is separate from the SWF file and the wrapper. For more information that is specific to QTP, see *Testing Adobe Flex Applications with HP QuickTest Professional*.

Using run-time loading

You can use the run-time testing files rather than compiling the automation libraries into your applications. This lets you test SWF files that have already been compiled without automated testing support. It also helps keep the SWF file size small. To do this, you use a SWF file that *does* include the automated testing libraries. In that SWF file, the SWFLoader class loads your application's SWF file which does not include the testing libraries. The result is that you can test the target SWF file in a testing tool such as QTP, even though the application SWF file was not compiled with automated testing support.

Flash Builder includes the following files necessary for run-time loading in the *flash_builder_install_dir*/sdks/4.6.0/templates/automation-runtimeloading-files directory:

- RunTimeLoading.html — The HTML wrapper that loads the run-time loader SWF file. This template includes code that converts the `automationswfurl` query string parameter to a `flashVars` variable that it passes to the application. You use this query string parameter to specify the name of the application you want to load and test.
- runtimeloading.mxml — The source code for the runtimeloading.swf file that you compile. The SWF file acts as a wrapper for your application. This SWF file includes the testing libraries so that you do not have to compile them into your application SWF file.

To use run-time loading:

1 Compile the runtimeloading.swf application from the runtimeloading.mxml file. You can use the batch file in the *flash_builder_install_dir*/sdks/4.6.0/templates/automation-runtimeloading-files directory. Execute this batch file from the sdks/4.6.0/frameworks directory. This batch file ensures that your runtimeloading.swf file includes the automation SWC files. You might need to add or remove SWC files to this file, depending on what features your application uses.

2 Deploy the runtimeloading.swf, RunTimeLoading.html, and your application's SWF file to a web server.

3 Request the RunTimeLoading.html file and pass the name of your SWF file as the value to the `automationswfurl` query string parameter. For example:

```
http://localhost/RunTimeLoading.html?automationswfurl=MyApp.swf
```

You can also create a custom HTML wrapper to use with the run-time loading feature, but it must use proper object naming. If you are using Flash Builder, you can generate a wrapper automatically. If you are using the SDK, you can use the wrapper template in the *flex_sdk*/templates/swfobject directory to create a wrapper for your application. When using a wrapper, the value of the `id` attributes can not contain any periods or hyphens.

If you want to recompile the runtimeloading.swf file without the batch file, be sure to include automated testing support by adding the appropriate automation SWC files with the `include-libraries` compiler option.

The batch file for compiling the runtimeloading.swf file is Windows only. To compile the SWF file on Mac OS or Linux, you can use the command line compiler or write your own batch file.

Creating test-friendly applications

As a Flex developer, there are some techniques that you can employ to make applications as “test friendly” as possible. One of the most important tasks that you can perform is to make sure that objects are identifiable in the testing tool’s scripts. This means that you should explicitly set the value of the `id` property or whatever property the testing tool uses to identify the object. Also be sure to use a meaningful string for that property so that the testing scripts are more readable. Finally, use unique IDs for each control so that the tool does not encounter ambiguous references.

Providing meaningful identification of objects

When working with testing tools such as QTP, a QC professional only sees the visual representation of objects in your application. A QC professional generally does not have access to the underlying code. When a QC professional records a script, it’s very helpful to see IDs that help the tester identify the object clearly. You should take some time to understand how testing tools interpret applications and determine what names to use for the test objects in the test scripts.

In most cases, testing tools use a visual cue, such as the label of a Button control, to identify the control in the script. Sometimes, however, testing tools use the Flex `id` property of an MXML tag to identify an object in the test script; if there is no value for the `id` property, testing tools use other properties, such as the `childIndex` property. In addition, the automation APIs provide an `automationName` property that you can use to explicitly set the ID of an object as it appears in the tool’s scripts.

You should give all testable MXML components an ID to ensure that the test script has a unique identifier to use when referring to that Flex control. You should also try to make these identifiers as human-readable as possible to make it easier for a QC professional to identify that object in the testing script. For example, set the `id` property of a Panel container inside a TabNavigator to *submit_panel* rather than *myPanel* or *p1*.

In some cases, agents do not use the `id` property, but it is a good practice to include it to avoid naming collisions or confusion. For more information about how QTP identifies Flex objects, see *Testing Adobe Flex Applications with HP QuickTest Professional*.

You should also set the value of the `automationName` property for all objects that are part of the application’s test. The value of this property appears in the testing scripts. Providing a meaningful name makes it easier for QC professionals to identify that object. For more information about using the `automationName` property, see “Setting the automationName property” on page 2289.

Avoiding renaming objects

Automation agents rely on the fact that some object’s properties should not be changed during run time. If you change the application property that is used by the agent as the object name at run time, unexpected results can occur.

For example, if you create a Button control without an `automationName` property, and you do not set the value of its `label` property during initialization, and then later set the value of the `label` property, the agent might get confused. This is because agents often use the value of the `label` property of a Button control to identify it in its object repository if the `automationName` property is not set. If you later set the value of the `label` property, or change the value of an existing `label` while the QC professional is recording a test, an automation tool such as QTP will create a new object in its repository instead of using the exising reference.

As a result, you should try to understand what properties are used to identify objects in the agent, and try to avoid changing those properties at run time. You should set unique, human-readable `id` or `automationName` properties for all objects that are included in the recorded script.

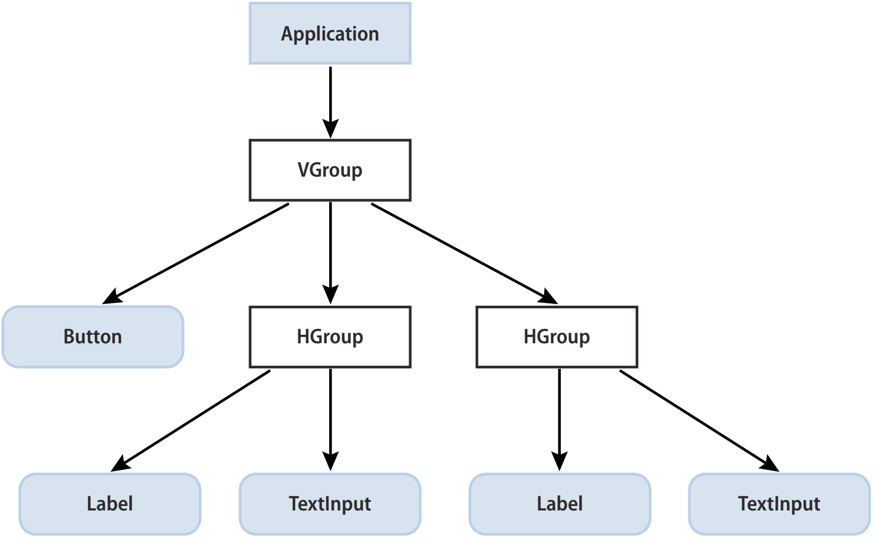
Coding containers

Containers are different from other kinds of controls because they are used both to record user interactions (such as when a user moves to the next pane in an Accordion container) and to provide layout logic.

Adding and removing containers from the automation hierarchy

In general, the automated testing tools reduce the amount of detail about nested containers in their scripts. They remove containers that have no impact on the results of the test or on the identification of the controls from the script. This applies to containers that are used exclusively for layout, such as the HGroup, VGroup and Canvas containers.

The following image shows the layout for a simple form-based application. The components with gray backgrounds appear in the automation tool's scripts because they can be interacted with by the user. Components with white backgrounds do not appear in the tool's scripts because they only provide layout logic.



Automation flowchart

In some cases, containers that are being used in multiple-view navigator containers, such as the ViewStack, TabNavigator or Accordion containers, do appear in the automation hierarchy. In these cases, they are added to the automation hierarchy to provide navigation.

Many composite components use containers, such as VGroup or HGroup, to organize their children. These containers do not have any visible impact on the application. So, these containers are usually excluded from the test scripts because there is no user interaction and no visual need for their operations to be recordable. By excluding a container from being tested, the test scripts are shorter and more readable.

To exclude a container from being recorded (but not exclude its children), set the container's `showInAutomationHierarchy` property to `false`. This property is defined by the UIComponent class, so all containers that subclass UIComponent have this property. Children of containers that are not visible in the hierarchy appear as children of the next highest visible parent. You can also use this property to exclude controls that you want to omit from testing, such as a Help button or decorative object.

The default value of the `showInAutomationHierarchy` property depends on the type of container. For containers that have visual elements or support user interaction, such as lists, Panel, Accordion, Application, DividedBox, and Form, the default value is `true`. For containers used exclusively for layout, such as Group, Canvas, Box, and FormItem, the default value is `false`.

The following example forces the VGroup containers to be included in the test script's hierarchy:

```
<?xml version="1.0"?>
<!-- agent/NestedButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:Panel title="ComboBox Control Example">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <s:HGroup id="hg">
            <s:VGroup id="vg1" showInAutomationHierarchy="true">
                <mx:Canvas id="c1">
                    <s:Button id="b1"
                        automationName="Nested Button 1"
                        label="Click Me"/>
                </mx:Canvas>
            </s:VGroup>
            <s:VGroup id="vg2" showInAutomationHierarchy="true">
                <mx:Canvas id="c2">
                    <s:Button id="b2"
                        automationName="Nested Button 2"
                        label="Click Me 2"/>
                </mx:Canvas>
            </s:VGroup>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

For additional information, see More about showInAutomationHierarchy.

Working with multiview containers

You should avoid using the same label on multiple tabs in multiview containers, such as TabNavigator and Accordion containers. Although the compiler allows you to use the same labels for each view, this is generally not an acceptable UI design practice and can cause problems with control identification in your testing environment. QTP, for example, uses the `label` properties of multiview containers to identify those views to testers. When two labels are the same, QTP uses different strategies to uniquely identify the tabs, which can result in a confusing name list.

Also, dynamically adding children to multiview containers can cause delays that might confuse the testing tool. You should try to avoid this.

Testing sub-applications

When testing applications that load sub-applications, you should be aware of the different types of applications that a main application can load. These include applications that:

- Were compiled with the same version of the compiler as the main application (single-versioned applications).
- Were compiled with different versions of the compiler in different application domains (multi-versioned applications).
- Are loaded into different security domains (sandboxed applications). These applications can be multi-versioned.

When testing applications that load other applications, there is a single point of communication between the automation tool and the applications built with Flex. This point is the main application. It acts as a gateway between the sub-applications and the automation tool. All events dispatched by the sub-applications are routed through the main application to the tool. And conversely, all calls from the tool are sent to the main application. The main application is responsible for routing those messages to the sub-applications.

When creating applications that will be tested, compile each sub-application and the main application with its own tool library, automation manager, and delegates. In other words, compile the application against the automation libraries.

When recording an application that loads sub-applications, the tool typically uses the IAutomationManager2's `getUniqueApplicationId()` method. This method returns a unique ID for each application. The resulting scripts contain longer strings to identify objects, but you can see the heirarchy of the applications in them. For example, a Spark Button control in a sub-application might be referenced as follows:

```
FlexApplication("loader1").FlexApplication("local2.swf").SparkButton("Submit Form");
```

A script that was written for the main application can be reused within a multi-application environment. You might be required to provide the application ID to the operations so that the target application can be uniquely identified.

Support for sub-application testing is built into your applications by default. As a developer, you do not need to add custom code to ensure that multi-application tests work. However, if you write a custom agent for a multi-application environment, keep the following in mind:

- Multi-versioned applications should use the sandbox root application to dispatch events. This is required for events that are communicated across application boundaries. You can use the following code to get a reference to the sandbox root:

  ```
  private  var sandboxRoot:IEventDispatcher;
  var sm:ISystemManager = Application.application.systemManager;
  sandboxRoot = sm.getSandboxRoot();
  ```

- Untrusted (or sandboxed) applications use the SWFBridge to communicate across security domain boundaries. Each application has a SWFBridge. The SWFBridge has a child that corresponds to the sub-application that was loaded with a SWFLoader in that application.

Writing the wrapper

In most cases, the testing tool requests a file from a web server that embeds the application. This file, known as the *wrapper*, is often written in HTML, but can also be a JSP, ASP, or other file that browsers interpret. You can request the SWF file directly in the testing tool by using the file protocol, but then you must ensure that the SWF file is trusted.

If you are using Adobe LiveCycle Data Services ES or Flash Builder, you can generate a wrapper automatically. If you are using the Flex SDK, you can use the *flex_sdk*/templates/swfobject/index.template.html file as a basis to create a wrapper for your application.

When you careate a custom wrapper, your wrapper's `<object>` tag must have an `id` attribute, and the value of the `id` attribute can not contain any periods or hyphens. The convention is to set the `id` to match the name of the root MXML file in the application.

When you use the SWFObject 2 wrapper as a template, you must set the id of the application on the `attributes.id` property.

When you use Flash Builder to generate a wrapper, the value of the `id` attribute is the name of the root application file. You do not have to make any changes to this attribute.

When you generate a wrapper with the LiveCycle Data Services ES server, the object tag's `id` attribute is valid. The following example shows the default object tag for a file named MainApp.swf:

```
<object id='MainApp' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab' height='600'
width='600'>
```

You are not required to change the value of the name in the `<embed>` tag because `<embed>` is used by Netscape-based browsers that do not support the testing feature. The `<object>` tag is used by Microsoft Internet Explorer.

Ensure that the object tag's `id` attribute is the same in the `<script>` and the `<noscript>` blocks of the wrapper.

Understanding the automation framework

The automation interfaces and the flow of the automation framework change as you initialize, record, and play back an automatable event.

About the automation interfaces

The Flex class hierarchy includes the following interfaces in the mx.automation.* package that enable automation:

Interface	Description
IAutomationClass and IAutomationClass2	Defines the interface for a component class descriptor.
IAutomationEnvironment	Provides information about the objects and properties of automatable components needed for communicating with agents.
IAutomationEventDescriptor	Defines the interface for an event descriptor.
IAutomationManager and IAutomationManager2	Defines the interface expected from an AutomationManager by the automation module.
IAutomationMethodDescriptor	Defines the interface for a method descriptor.
IAutomationMouseSimulator	Describes an object that simulates mouse movement so that components capturing the mouse use the simulated versions of the mouse cursor instead of the live Flash Player version.
IAutomationObject	Defines the interface for a delegate object implementing automation for a component.
IAutomationObjectHelper	Provides helper methods for the IAutomationObject interface.
IAutomationPropertyDescriptor	Describes a property of a test object as well as properties of an event object.
IAutomationTabularData	Defines the interface for components that provide their content information in a tabular form.

For more information about each of these interfaces, see the class's description in the ActionScript 3.0 Reference for the Adobe Flash Platform.

About the IAutomationObjectHelper interface

The IAutomationObjectHelper interface helps the components accomplish the following tasks:

- Replay mouse and keyboard events; the helper generates proper sequence of player level mouse and key events.
- Generate AutomationIDPart for a child: AutomationIDPart would be requested by the Automation for representing a component instance to agents.
- Find a child matching a AutomationIDPart: Automation would request the component to locate a child matching the AutomationIDPart supplied by an agent to it.
- Avoid synchronization issues: Agents invoke methods on Automation requesting operations on components in a sequence. Components may not be ready all the time to perform operations.

For example, an agent can invoke `comboBox.Open`, `comboBox.select "Item1"` operations in a sequence. Because it takes time for the drop-down list to open and initialize, it is not possible to run the select operation immediately. You can place a wait request during the open operation execution. The wait request should provide a function for automation, which can be invoked to check the ComboBox control's readiness before invoking the next operation.

Automated testing workflow with the QTP automation tool

Before you automate custom components, you might find it helpful to see the order of events during which Flex's automation framework initializes, records, and plays back events with a tool such as QTP. You should keep in mind that the QTP adapter class' implementation is only one way to use the automation API for automated testing.

Automated testing initialization

1. The user launches the application. Automation initialization code associates component delegate classes with component classes. Component delegate classes implement the IAutomationObject interface.
2. AutomationManager is a mixin. Its instance is created in the mixin `init()` method.
3. The SystemManager initializes the application. Component instances and their corresponding delegate instances are created. Delegate instances add event listeners for events of interest.
4. QTPAgent class is a mixin. In its `init()` method, it registers itself for the `FlexEvent.APPLICATION_COMPLETE` event which is dispatched from the SystemManager. On receiving the event, it creates a QTPAdapter object.
5. QTPAdapter sets up the ExternalInterface function map. QTPAdapter loads the QTP Plugin DLLs by creating the ActiveX object to communicate with QTP.
6. The QTPAdapter requests the XML environment information from the plugin and passes it to the AutomationManager.
7. The XML information is stored in a chain of AutomationClass, AutomationMethodDescriptor, and AutomationPropertyDescriptor objects.

Automated testing recording

1. The user clicks the Record button in QTP.
2. QTP calls the `QTPAdapter.beginRecording()` method. QTPAdapter adds a listener for `AutomationRecordEvent.RECORD` from the AutomationManager.
3. The QTPAdapter notifies AutomationManager about this by calling the `beginRecording()` method. The AutomationManager adds a listener for the `AutomationRecordEvent.RECORD` event from the SystemManager.
4. The user interacts with the application. In this example, suppose the user clicks a Button control.
5. The `ButtonDelegate.clickEventHandler()` method dispatches an AutomationRecordEvent event with the `click` event and Button instance as properties.

6 The AutomationManager `record` event handler determines which properties of the `click` event to store, based on the XML environment information. It converts the values into proper type or format. It dispatches the `record` event.

7 The QTPAdapter event handler receives the event. It calls the `AutomationManager.createID()` method to create the AutomationID object of the button. This object provides a structure for object identification.

The AutomationID structure is an array of AutomationIDParts. An AutomationIDPart is created by using IAutomationObject. (The `UIComponent.id`, `automationName`, `automationValue`, `childIndex`, and `label` properties of the Button control are read and stored in the object. The `label` property is used because the XML information specifies that this property can be used for identification for the Button.)

8 The QTPAdapter uses the `AutomationManager.getParent()` method to get the logical parent of the Button control. The AutomationIDPart objects of parent controls are collected at each level up to the application level.

9 All these AutomationIDParts are made part of an AutomationID object.

10 The QTPAdapter sends the information in a call to QTP.

11 At this point, QTP might call the `AutomationManager.getProperties()` method to get the property values of the Button control. The property type information and codec that should be used to modify the value format are gotten from the AutomationPropertyDescriptor.

12 User stops recording. This is propagated by a call to the `QTPAdapter.endRecording()` method.

Automated testing playback

1 The user clicks the Playback button in QTP.

2 The `QTPAdapter.findObject()` method is called to determine whether the object on which the event has to be played back can be found. The AutomationID object is built from the XML data received. The `AutomationManager.resolveIDToSingleObject()` method is invoked to see if QTP can find one unique object matching the AutomationID. The `AutomationManager.getChildren()` method is invoked from application level to find the child object. The `IAutomationObject.numAutomationChildren` property and the `IAutomationObject.getAutomationChildAt()` method are used to navigate the application.

3 The `AutomationManager.isSynchronized()` and `AutomationManager.isVisible()` methods ensure that the object is fully initialized and is visible so that it can receive the event.

4 QTP invokes the `QTPAdpater.run()` method to play back the event. The `AutomationManager.replayAutomatableEvent()` method is called to replay the event.

5 The AutomationMethodDescriptor for the `click` event on the Button is used to copy the property values (if any).

6 The `AutomationManager.replayAutomatableEvent()` method invokes the `IAutomationObject.replayAutomatableEvent()` method on the delegate class. The delegate uses the `IAutomationObjectHelper.replayMouseEvent()` method (or one of the other replay methods, such as `replayKeyboardEvent()`) to play back the event.

7 If there are check points recorded in QTP, the `AutomationManager.getProperties()` method is invoked to verify the values.

Instrumenting events

When you extend components that are already instrumented, you do not have to change anything to ensure that those components' events can be recorded by a testing tool. For example, if you extend a Button class, the class still dispatches the automation events when the Button is clicked, unless you override the Button control's default event dispatching behavior.

Automation events (sometimes known in automation tools such as QTP as *operations*) are not the same as Flex events. Flex must dispatch an automation event as a separate action. Flex dispatches them at the same time as Flex events, and uses the same event classes, but you must decide whether to make a Flex event visible to the automation tool.

Not all events on a control are instrumented. You can instrument additional events by using the instructions in "Instrumenting existing events" on page 2281.

If you change the instrumentation of a component, you might also be required to edit that component's entry in the class definitions file. This is described in "Using the class definitions file" on page 2286.

Instrumenting existing events

Events have different levels of relevance for the QC professional. For example, a QC professional is generally interested in recording and playing back a `click` event on a Button control. The QC professional is not generally interested in recording all the events that occur when a user clicks the Button, such as the `mouseOver`, `mouseDown`, `mouseUp`, and `mouseOut` events. For this reason, when a tester clicks on a Button control with the mouse, testing tools only record and play back the `click` event for the Button control and not the other, lower-level events.

There are some circumstances where you would want to record events that are normally ignored by the testing tool. But the testing tool's object model only records events that represent the end-user's gesture (such as a click or a drag and drop). This makes a script more readable and it also makes the script robust enough so that it does not fail if you change the application slightly. So, you should carefully consider whether to add a new event to be tested or rely on events in the existing object model.

You can see a list of events that the QTP automation tool can record for each component in the *QTP Object Type Information* (or *class definition*) documents. The MX Button control, for example, supports the following operations, based on its entry in the TEAFlex.xml file:

- `ChangeFocus`
- `Click`
- `MouseMove`
- `SetFocus`
- `Type`

All of these operations except for `MouseMove` are automatically recorded by QTP by default. The QC professional must explicitly add the `MouseMove` operation to their QTP script for QTP to record play back the related event.

However, you can alter the behavior of your application so that this event is recorded by the testing tool. To add a new event to be tested, you override the `replayAutomatableEvent()` method of the IAutomationObject interface. Because the UIComponent class implements this interface, all subclasses of UIComponent (which include all visible Flex controls) can override this method. To override the `replayAutomatableEvent()` method, you create a custom class, and override the method in that class.

The `replayAutomatableEvent()` method has the following signature:

```
public function replayAutomatableEvent(event:Event):Boolean
```

The *event* argument is the Event object that is being dispatched. In general, you pass the Event object that triggered the event. Where possible, you pass the specific event, such as a MouseEvent, rather than the generic Event object.

The following example shows a custom Spark Button control that overrides the `replayAutomatableEvent()` method. This method checks for the `mouseMove` event and calls the `replayMouseEvent()` method if it finds that event. Otherwise, it calls its superclass' `replayAutomatableEvent()` method.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- agent/CustomButton.mxml -->
<s:Button xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
        <fx:Script>
        <![CDATA[
            import flash.events.Event;
            import flash.events.MouseEvent;
            import mx.automation.Automation;
            import mx.automation.IAutomationObjectHelper;
            override public function
                replayAutomatableEvent(event:Event):Boolean {
                trace('in replayAutomatableEvent()');
                var help:IAutomationObjectHelper = Automation.automationObjectHelper;

                if (event is MouseEvent && event.type == MouseEvent.MOUSE_MOVE) {
                    return help.replayMouseEvent(this, MouseEvent(event));
                } else {
                    return super.replayAutomatableEvent(event);
                }
            }
        ]]>
    </fx:Script>
</s:Button>
```

In the application, you call the AutomationManager's `recordAutomatableEvent()` method when the user moves the mouse over the button. The following application uses this custom class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- agent/CustomButtonApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:ns1="*" initialize="doInit()">
    <fx:Script>
        <![CDATA[
            import mx.automation.*;
            public function doInit():void {
                b1.addEventListener(MouseEvent.MOUSE_MOVE, dispatchLowLevelEvent);
            }
            public function dispatchLowLevelEvent(e:MouseEvent):void {
                var help:IAutomationManager = Automation.automationManager;
                help.recordAutomatableEvent(b1,e,false);
            }
        ]]>
    </fx:Script>
    <ns1:CustomButton id="b1"
        toolTip="Mouse moved over"
        label="CustomButton"/>

</s:Application>
```

If the event is not one that is currently recordable, you also must define the new event for the agent. Typically, you do this by adding a new entry to a class definitions file. For a tool such as QTP, the class definitions are in the TEAFlex.xml file. Automation tools can use files like this to define the events, properties, and arguments for each class of test object. For more information, see “Instrumenting events” on page 2280. If you wanted to add support for the mouseOver event in QTP, for example, you add the following to the SparkButton object’s entry in the TEAFlex.xml file:

```
<Operation Name="MouseOver" PropertyType="Method" ExposureLevel="CommonUsed">
    <Implementation Class="flash.events::MouseEvent" Type="mouseOver"/>
        <Argument Name="keyModifier" IsMandatory="false" DefaultValue="0">
            <Type VariantType="Enumeration"
                ListOfValuesName="FlexKeyModifierValues" Codec="keyModifier"/>
            <Description>Occurs when the user moves mouse over the component</Description>
        </Argument>
</Operation>
```

In the preceding example, however, the mouseMove event is already in the SparkButton object’s entry in that file, so no editing is necessary. The difference now is that the QC professional does not have to explicitly add the event to their script. After you compile this application and deploy the new TEAFlex.xml file to the QTP testing environment, QTP records the mouseMove event for all of the CustomButton objects.

For more information about the class definitions file, see “Using the class definitions file” on page 2286.

Instrumenting custom events

When you extend components, you often add events that are triggered by new functionality of that component. You can instrument custom events by adding a call to the Automation.automationManager2 class’s recordCustomAutomationEvent() method. You usually do this in the same place that you call the dispatchEvent() method. You do not replace the call to the dispatchEvent() method.

The following example from a custom event class builds a new AutomationRecordEvent and calls the recordCustomAutomationEvent() method to ensure that it is recorded:

```
this.addEventListener(MyComponentEvent.myEvent, myHandler);
private function myHandler(event:MyComponentEvent) {
    var eventToRecord:AutomationRecordEvent =
        new AutomationRecordEvent(AutomationRecordEvent.CUSTOM_RECORD);
    eventToRecord.automationObject = this;
     // Provide the name of the event:
    eventToRecord.name = "MyCustomEventName";
    // Provide the details to be recorded. This should be of basic data types.
    eventToRecord.args = [MySelectionDetails];
    Automation.automationManager2.recordCustomAutomationEvent(eventToRecord);
}
```

For a tool such as QTP to recognize the event as a recordable operation, you must also add the event to the control’s entry in the class definitions file (TEAFlex.xml).

To replay the custom event, you listen for the AutomationCustomReplayEvent, get the details about it, and replay the event, as the following example shows:

```
Automation.automationManager2.addEventListener(AutomationCustomReplayEvent.Replay,
    handleReplay,false, EventPriority.DEFAULT+1);
private function handleReplay(event:AutomationCustomReplayEvent):void {
    if (event.automationObject  == this) {
        // take the name and args:
        var name:String = event.name;
        var args:Array = event.args;
        // Use the above do the required replay:
        event.preventDefault(); // prevent the default replay
    }
}
```

Blocking and overriding events

In some cases, you might want to block or override the default events that are recorded for a component. You might even want to replace them with a different event. This is often the case with events dispatched from custom components.

For example, suppose you have an application that uses a custom component that consists of a List and a Button control. When the user selects an item from the list, the application records a `change` event. However, if you change the List control to a RadioButtonGroup in your custom component, you no longer want to record the `change` event, but rather, you want to record the `itemClick` event.

The result is that automation scripts can break due to changes in the custom component. To get around this, you can block the events and dispatch a higher-level event that corresponds to the user's gesture rather than the specific event.

To block the recording of the default events, create an event listener that listens for all `AutomationRecordEvent.RECORD` events. Set its priority to be higher than other listeners. In the handler, call the event's `preventDefault()` method, as the following example shows:

```
Automation.automationManager2.addEventListener(AutomationRecordEvent.RECORD,
    blockEvents,false, EventPriority.DEFAULT+1);
private function blockEvent(event:AutomationRecordEvent):void {
    // Block all events on list1 and button1:
    if ((event.automationObject == list1) || (if (event.automationObject == button1)) {
        event.preventDefault();
    }
}
```

You can then dispatch the more generic event that resembles the user's gesture as shown in "Instrumenting custom events" on page 2283.

Instrumenting custom components

The process of creating a custom component that supports automated testing is called instrumentation. Flex framework components are instrumented by attaching a delegate class to each component at run time. The *delegate class* defines the methods and properties required to perform instrumentation.

If you extend an existing component that is instrumented, such as a Button control, you inherit its parent's instrumentation, and are not required to do anything else to make that component testable. If you create a component that inherits from UIComponent, you must instrument that class in one of the following ways:

- Create a delegate class that implements the required interfaces.
- Add testing-related code to the component.

You usually instrument components by creating delegate classes. You can also instrument components by adding automation code inside the components, but this is not a recommended practice. It creates tighter coupling between automated testing code and component code, and it forces the automated testing code to be included in a production SWF file.

In both methods of instrumenting a component, you must specify any new events to the agent. With QTP, for example, you must add your new component's information to the class definitions file so that QTP recognizes that component. For more information about this file, see "Using the class definitions file" on page 2286.

Consider the following additional factors when you instrument custom components:

- Composition. When instrumenting components, you must consider whether the component is a simple component or a composite component. Composite components are components made up of several other components. For example, a TitleWindow that contains form elements is a composite component.
- Container hierarchy. You should understand how containers are viewed in the automation hierarchy so that the QC professional can easily test the components. Also, you should be aware that you can manipulate the hierarchy to better suit your application by setting some automation-related properties.
- Automation names. Custom components sometimes have ambiguous or unclear default automation names. The ambiguous names make it more difficult in automation tools to determine what component a script is referring to. Component authors can manually set the value of the `automationName` property for all components except item renderers. For item renderers, use the `automationValue`.

Creating a delegate class

To instrument custom components with a delegate, you must do the following:

- Create a delegate class that implements the required interfaces. In most cases, you extend the UIComponentAutomationImpl class. You can instrument any component that implements IUIComponent.
- Register the delegate class with the AutomationManager.
- Define the component in a class definitions XML file.

The delegate class is a separate class that is not embedded in the component code. This helps to reduce the component class size and also keeps automated testing code out of the final production SWF file.

All visual Flex controls have their own delegate classes. These classes are in the spark.automation.delegates.components.* package for Spark controls, and the mx.automation.delegates.* package for MX components. The class names follow a pattern of *Class_name*AutomationImpl. For example, the delegate class for the Spark Button control is spark.automation.delegates.components.SparkButtonAutomationImpl. The delegate class for the MX Button control is mx.automation.delegates.controls.ButtonAutomationImpl.

The delegate class defines the following:

- The `enableAutomation()` method
- `The replayAutomatableEvent()` method
- Event listeners

All subclasses of UIComponent store a reference to their delegate in the `automationDelegate` property. This behavior is defined in the UIComponent initializer, which calls the delegate's `enableAutomation()` method. If you create a custom component that does not subclass UIComponent, then you must manually call the delegate's `enableAutomation()` method in your custom component's initilization code.

You map the delegate's unique name and value properties to the `automationName` and `automationValue` properties.

Instrument with a delegate class

1. Create a delegate class.
2. Mark the delegate class as a mixin by using the `[Mixin]` metadata keyword.
3. Register the delegate with the AutomationManager by calling the `Automation.registerDelegateClass()` method in the `init()` method. The following code is a simple example:

```
[Mixin]
public class MyCompDelegate {
    public static init(root:DisplayObject):void {
        // Pass the component and delegate class information.
        Automation.registerDelegateClass(MyComp, MyCompDelegate);
    }
}
```

   You pass the custom class and the delegate class to the `registerDelegateClass()` method.

4. Add the following code to your delegate class:
   a. Override the getter for the `automationName` property and define its value. This is the name of the object as it usually appears in automation tools such as QTP. If you are defining an item renderer, use the `automationValue` property instead.
   b. In the constructor, add event listeners for events that the automation tool records.
   c. Override the `replayAutomatableEvent()` method. The AutomationManager calls this method for replaying events. In this method, return whether the replay was successful. You can use methods of the helper classes to replay common events.

      For examples of delegates, see the source code for the Flex controls in the spark.automation.delegates.components.* package.

5. Link the delegate class with the application SWF file in one of these ways:
   - Add the following `includes` compiler option to link in the delegate class:

     ```
     mxmlc -includes MyCompDelegate -- FlexApp.mxml
     ```

   - Build a SWC file for the delegate class by using the compc component compiler:

     ```
     compc -source-path+=. -include-classes MyCompDelegate -output MyComp.swc
     ```

     Then include this SWC file with your application by using the following `include-libraries` compiler option:

     ```
     mxmlc -include-libraries MyComp.SWC -- FlexApp.mxml
     ```

     This approach is useful if you have many components and delegate classes and want to include them as a single library so that you can share them with other developers.

6. After you compile your application with the new delegate class, you must define the new interaction for the agent and the automation tool. For QTP, you must add the new component to QTP's custom class definition XML file. For more information, see "Using the class definitions file" on page 2286.

Using the class definitions file

The class definitions file contains information about all instrumented components. This file provides information about the components to the automation agent, including what events can be recorded and played back, the name of the component, and the properties that can be tested.

An example of a class definitions file is the TEAFlex.xml file, which is specific to the QTP automation tool. This file is included in the Flex Automation Package.

The TEAFlex.xml file is located in the "*QTP_plugin_install*/Adobe Flex 4 Plug-in for HP QuickTest Pro" directory. QTP recognizes any file in that directory that matches the pattern TEAFlex*.xml, where * can be any string. This directory also contains a TEAFlexCustom.xml file that you can use as a starting point for adding custom component definitions.

The TEAFlex.xml and FlexEnv.xml class definitions files describe instrumented components with the following basic structure:

```
<TypeInformation>
    <ClassInfo>
        <Description/>
        <Implementation/>
        <TypeInfo>
            <Operation/>
            ...
        </TypeInfo>
        <Properties>
            <Property/>
            ...
        </Properties>
    </ClassInfo>
</TypeInformation>
```

The top level tag is `<TypeInformation>`. You define a new class that uses the `<ClassInfo>` tag, which is a child tag of the `<TypeInformation>` tag. The `<ClassInfo>` tag has child tags that further define the instrumented classes. The following table describes these tags:

Tag	Description
ClassInfo	Defines the class that is instrumented, for example, SparkButton. This is the name that the automation tools use for the Spark Button control. (For the MX Button control, the name is FlexButton.) Attributes of this tag include `Name`, `GenericTypeID`, `Extends`, and `SupportsTabularData`.
Description	Defines the text that appears in the automation tool to define the component. This is not implemented in the AutoQuick example, but is implemented for QTP.
Implementation	Defines the class name, as it is known by the Flex compiler, for example, Button or MyComponent. There is an optional property of the `<Implementation>` element, `version`. You use the `version` property to differentiate controls that are in same package and have the same name, but have differences in their API across the versions. For example: `<Implementation Class="my.custom.controls::BigButton" version="1.5"/>`
TypeInfo	Defines events for this class. Each event is defined in an `<Operation>` child tag, which has two child tags: • The `<Implementation>` child tag associates the operation with the actual event. • Each operation can also define properties of the event object by using an `<Argument>` child tag.
Properties	Defines properties of the class. Each property is defined in a `<Property>` child tag. Inside this tag, you define the property's type, name, and description. For each `Property`, if the `ForDescription` attribute is `true`, the property is used to uniquely identify a component instance in the automation tool; for example, the `label` property of a Button control. QTP lists this property as part of the object in QTP object repository. If the `ForVerfication` attribute is `true`, the property is visible in the properties dialog box in QTP. If the `ForDefaultVerification` tag is `true`, the property appears selected by default in the dialog box in QTP. This results in verification of the property value in the checkpoint.

The following example adds a new component, MyComponent, to the class definition file. This component has one instrumented event, `click`:

```
<TypeInformation xsi:noNamespaceSchemaLocation="ClassesDefintions.xsd" Priority="0"
PackageName="TEA" Load="true" id="Flex" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ClassInfo Name="MyComponent" GenericTypeID="mycomponent"
        Extends="FlexObject" SupportsTabularData="false">
        <Description>FlexMyComponent</Description>
        <Implementation Class="MyComponent"/>
        <TypeInfo>
            <Operation Name="Select" PropertyType="Method"
                ExposureLevel="CommonUsed">
                <Implementation Class="myComponentClasses::MyComponentEvent" Type="click"/>
            </Operation>
        </TypeInfo>
        <Properties>
            <Property Name="automationClassName" ForDescription="true">
                <Type VariantType="String"/>
                <Description>This is MyComponent.</Description>
            </Property>
            <Property Name="automationName" ForDescription="true">
                <Type VariantType="String"/>
                <Description>The name used by tools to id an object.</Description>
            </Property>
            <Property Name="className" ForDescription="true">
                <Type VariantType="String"/>
                <Description>To be written.</Description>
            </Property>
            <Property Name="id" ForDescription="true" ForVerification="true">
                <Type VariantType="String"/>
                <Description>Developer-assigned ID.</Description>
            </Property>
            <Property Name="index" ForDescription="true">
                <Type VariantType="String"/>
                <Description>The index relative to its parent.</Description>
            </Property>
        </Properties>
    </ClassInfo>
    ...
</TypeInformation>
```

You can edit the class definitions file to add a new recordable event to an existing component. To do this, you insert a new `<Operation>` in the control’s `<TypeInfo>` block. This includes the implementation class of the event, and any arguments that the event might take.

The following example adds a new event, `MouseOver`, with several arguments to the Button control:

```
<TypeInfo>
    <Operation ExposureLevel="CommonUsed" Name="MouseOver" PropertyType="Method">
        <Implementation Class="flash.events::MouseEvent" Type="mouseOver"/>
        <Argument Name="inputType" IsMandatory="false"
            DefaultValue="mouse">
            <Type VariantType="String"/>
        </Argument>
        <Argument Name="shiftKey" IsMandatory="false" DefaultValue="false">
            <Type VariantType="Boolean"/>
        </Argument>
        <Argument Name="ctrlKey" IsMandatory="false" DefaultValue="false">
            <Type VariantType="Boolean"/>
        </Argument>
        <Argument Name="altKey" IsMandatory="false" DefaultValue="false">
            <Type VariantType="Boolean"/>
        </Argument>
    </Operation>
</TypeInfo>
```

When you finish editing the class definitions file, you must distribute the new file to the automation tool users. For QTP, users must copy this file manually to the "*QTP_plugin_install*\Adobe Flex 4 Plug-in for HP QuickTest Pro" directory. When you replace the class definitions file in the QTP environment, you must restart QTP.

Setting the automationName property

The `automationName` property defines the name of a component as it appears in testing scripts. The default value of this property varies depending on the type of component. For example, a Button control's `automationName` is the label of the Button control. Sometimes, the `automationName` is the same as the control's `id` property, but this is not always the case.

For some components, Flex sets the value of the `automationName` property to a recognizable attribute of that component. This helps QC professionals recognize that component in their scripts. For example, a Spark Button labeled "Process Form Now" appears in the testing scripts as `SparkButton("Process Form Now")`.

If you implement a new component, or subclass an existing component, you might want to override the default value of the `automationName` property. For example, UIComponent sets the value of the `automationName` to the component's `id` property by default, but some components use their own methods of setting its value.

The following example sets the `automationName` property of the ComboBox control to "Credit Card List"; rather than using the `id` property, the testing tool typically uses "Credit Card List" to identify the ComboBox in its scripts:

```
<?xml version="1.0"?>
<!-- agent/SimpleComboBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            [Bindable]
            public var cards: Array = [
                {label:"Visa", data:1},
                {label:"MasterCard", data:2},
                {label:"American Express", data:3}
            ];

            [Bindable]
            public var selectedItem:Object;
        ]]>
    </fx:Script>
    <s:Panel title="ComboBox Control Example">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>

        <mx:ComboBox id="cb1" dataProvider="{cards}" width="150"
            close="selectedItem=ComboBox(event.target).selectedItem"
            automationName="Credit Card List"/>
        <s:VGroup width="250">
            <s:Label width="200" color="blue"
                text="Select a type of credit card."/>
            <s:Label text="You selected: {selectedItem.label}"/>
            <s:Label text="Data: {selectedItem.data}"/>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

If you do not set the value of the `automationName` property, the name of an object that appears in a testing tool is sometimes a property that can change while the application runs. If you set the value of the `automationName` property, testing scripts use that value rather than the default value. For example, by default, QTP uses a Button control's `label` property as the name of the Button in the script. If the label changes, the script can break. You can prevent this from happening by explicitly setting the value of the `automationName` property.

Buttons that have no label, but have an icon, are recorded by their index number. In this case, you should ensure that you set the `automationName` property to something meaningful so that the QC professional can recognize the Button in the script. This might not be necessary if you set the `toolTip` property of the Button because some tools such as QTP use that value if there is no label.

After the value of the `automationName` property is set, you should not change the value during the component's life cycle.

For item renderers, use the `automationValue` property rather than the `automationName` property. You do this by overriding the `createAutomationIDPart()` method and returning a new value that you assign to the `automationName` property, as the following example shows:

```
<mx:List xmlns:mx="http://www.adobe.com/2006/mxml">
    <fx:Script>
        <![CDATA[
            import mx.automation.IAutomationObject;
            override public function
                createAutomationIDPart(item:IAutomationObject):Object {
                    var id:Object = super.createAutomationIDPart(item);
                    id["automationName"] = id["automationIndex"];
                    return id;
            }
        ]]>
    </fx:Script>
</mx:List>
```

This technique works for any container or list-like control to add index values to their children. There is no method for a child to specify an index for itself.

Instrumenting composite components

Composite components are custom components made up of two or more components. A common composite component is a form that contains several text fields, labels, and buttons. Composite components can be MXML files or ActionScript classes.

By default, you can record operations on all instrumented child controls of a container. If you have a Button control inside a custom TitleWindow container, the QA professional can record actions on that Button control just like on any Button control. You can, however, create a composite component in which some of the child controls are instrumented and some are not. To prevent the operations of a child component from being recorded, you override the following methods:

- `numAutomationChildren` getter
- `getAutomationChildAt()`

The `numAutomationChildren` property is a read-only property that stores the number of automatable children that a container has. This property is available on all containers that have delegate implementation classes. To exclude some children from being automated, you return a number that is less than the total number of children.

The `getAutomatedChildAt()` method returns the child at the specified index. When you override this method, you return null for the unwanted child at the specified index, but return the other children as you normally would.

The following custom composite component is written in ActionScript. It consists of a VGroup container with three buttons (OK, Cancel, and Help). You cannot record the operations of the Help button. You can record the operations of the other Button controls, OK and Cancel. The following example sets the values of the OK and Cancel buttons' `automationName` properties. This makes those button controls easier to recognize in the automated testing tool's scripts.

```
// agent/MyVGroup.as
package { // Empty package
    import mx.core.UIComponent;
    import spark.components.VGroup;
    import spark.components.Button;
    import mx.automation.IAutomationObject;
    import spark.automation.delegates.components.SparkGroupAutomationImpl;
    public class MyVGroup extends VGroup {
        public var btnOk : Button;
        public var btnHelp : Button;
        public var btnCancel : Button;
        public function MyVGroup():void { // Constructor
        }

        override protected function createChildren():void {
            super.createChildren();

            btnOk = new Button();
            btnOk.label = "OK";
            btnOk.automationName = "OK_custom_form";
            addElement(btnOk);

            btnCancel = new Button();
            btnCancel.label = "Cancel";
            btnCancel.automationName = "Cancel_custom_form";
            addElement(btnCancel);

            btnHelp = new Button();
            btnHelp.label = "Help";
            btnHelp.showInAutomationHierarchy = false;
            addElement(btnHelp);
        }

        override public function get numAutomationChildren():int {
            return 2; //instead of 3
        }
        override public function
            getAutomationChildAt(index:int):IAutomationObject {
            switch(index) {
                case 0:
                    return btnOk;
                case 1:
                    return btnCancel;
            }
            return null;
        }
    } // Class
} // Package
```

The following application uses the MyVGroup custom component:

```
<?xml version="1.0"?>
<!-- agent/NestedButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:comps="*">
    <s:Panel title="Composite VGroup with Custom Automation Settings">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <comps:MyVGroup/>
    </s:Panel>
</s:Application>
```

To make this solution more portable, you could create a custom Button control and add a property that determines whether a Button should be testable. You could then set the value of this property based on the Button instance (for example, `btnHelp.useInAutomation = false`), and check against it in the overridden `getAutomationChildAt()` method, before returning null or the button instance.

For better performance, you can use the `getAutomationChildren()` method rather than the `getAutomationChildAt()` method; for example:

```
var childList:Array = getAutomationChildren();
var n:int = childList ? childList.length : 0;
for (var i:int = 0; i < n; i++) {
    var child:IAutomationObject = childList[i];
    ...
}
```

Custom agents

The Automation Framework defines a single API that has two parts:

- Component API — Components must implement this API to support automation features. Developers can choose to put the code either in the main component itself or in a mixin class. Mixin classes implement this API for Flex components.
- Agent API — Agents use this API to communicate with the component API.

Component developers implement the component API for their component once and then the component is ready to converse with any agent. Agent developers implement the agent API for their specific feature or tool and it is able to work with any application built with Flex. For more information, see “About the automation APIs” on page 2294.

Uses for agents

You can use agents to gather metrics information, to use automated testing, and to run applications at different locations at the same time.

Metrics

You might want to analyze how your online applications are being used. By gathering metrics information, you can answer the following questions:

- What product views are most popular?
- When do users abandon the checkout process?
- What is a typical path through my application?

- How many product views does a user look at during a session?

Automated testing

Maintaining the quality of a large software application is difficult. Verifying lots of functionality in any individual build can take a QA engineer many hours or even days, and much of the work between builds is repetitive. To alleviate this difficulty, automated testing tools have been created that can use applications and verify behavior without human intervention. Major application environments such as Java and .NET have testing tool support from vendors such as QTP and Segue.

By using the Flex automation API, you can:

- Record and replay events in a separate tool
- Manage communication between components and agents

Co-browsing

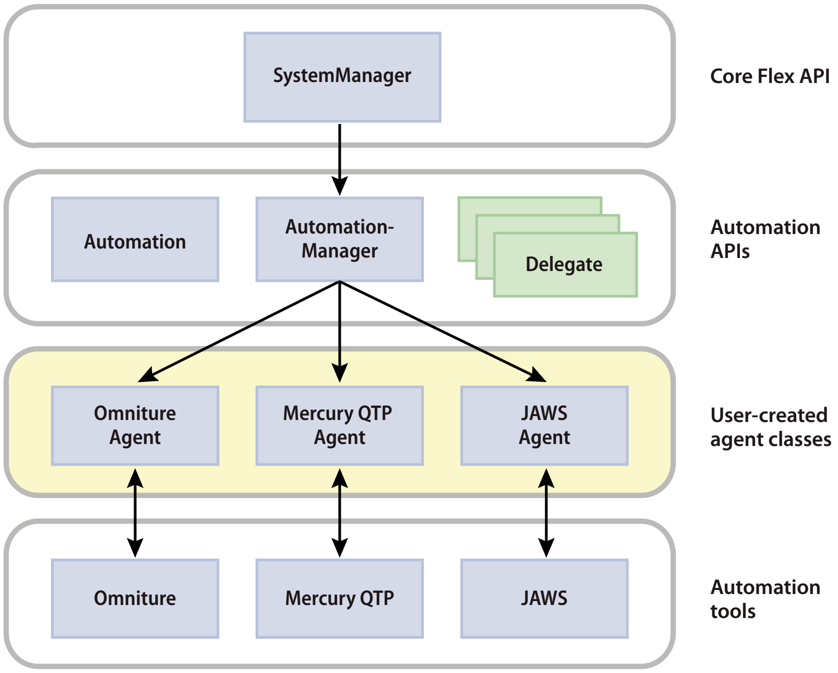
You might want to run the same application at different locations and view the application at the same time. By using the automation API, you can ensure that the applications are synchronized as users navigate through the them. User interaction at any location can be played at other locations and other users can see the action in real time.

About the automation APIs

There are four main parts that enable the automation framework in Flex:

- Core Flex API
- Automation APIs
- Agent class (also known as an adapter)
- Automation tools

The following illustration shows the relationship between these parts:



About the SystemManager class

The SystemManager class is one of the highest level classes in an application built with Flex. Every application built with Flex has a SystemManager class. It is the parent of all displayable objects in the application, such as the main spark.components.Application instance and all pop-up windows, ToolTip instances, cursors, and so on.

SystemManager calls the `init()` method on all mixins. The SystemManager class is responsible for creating the AutomationManager class as well as the Agent and the delegate classes. The SystemManager class is also responsible for adding the Application object to Adobe® Flash® Player or Adobe® AIR™ stage (root).

About the AutomationManager class

The AutomationManager class is a Singleton that extends the EventDispatcher class. It implements the IAutomationManager, IAutomationObjectHelper, and IAutomationMouseSimulator interfaces.

AutomationManager is a mixin, so its `init()` method is called by the SystemManager class when the application is initialized. In the `init()` method, the AutomationManager class adds an event listener for the `Event.ADDED` event. This event is dispatched by Flash Player or AIR whenever a display object is added to the display list. When that happens, Flash Player or AIR calls the `childAddedHandler()` method:

```
root.addEventListener(Event.ADDED, childAddedHandler, false, 0, true);
```

In the `childAddedHandler()` method, the AutomationManager class:

- Creates a new instance of a delegate for each display object.
- Adds the display object to a delegate class map. This maps the display object to a delegate instance; for example:

  ```
  Automation.delegateDictionary[componentClass] = Automation.delegateDictionary[className];
  ```

  The delegate class map is a property of the Automation class.
- Ensures that all children of the new display object are added to the class map as well.

In the `recordAutomatableEvent()` method, the AutomationManager:

- Creates an `AutomationRecordEvent.RECORD` event.
- Dispatches the `RECORD` events. The agent listens for these events.

The `recordAutomatableEvent()` method is called by the delegates.

About the Automation class

During application initialization, an instance of the Automation class is created. This is a static class that maintains a map of its delegate class to its component class.

This object does the following for recording events:

- Provides access to the AutomationManager class
- Creates a `delegateDictionary` as a static property

For example, there is a MyButton class that extends the Button class, but it does not have its own delegate class (MyButton might not add any new functionality to be recorded or played back). When the AutomationManager encounters an instance of the MyButton class, it checks with the Automation class for a corresponding delegate class. When it fails to find one, it uses the `getSuperClassName()` method to get the super class of the MyButton class, which is the Button class.

The AutomationManager then tries to find the delegate for this Button class. At that point, the AutomationManager adds a new entry into the delegate-component class for the MyButton class, associating it with the ButtonDelegateAutomationImpl class, so that next time the AutomationManager can find this mapping without searching the inheritance hierarchy.

About the delegate classes

The delegate classes provide automation hooks to the Flex components. The delegate classes are in the spark.automation.delegates.* and mx.automation.delegates.* packages. They extend the UIComponentAutomationImpl class. The delegates for the Spark components are named Spark*ControlName*AutomationImpl. The delegates for the MX components are named *ControlName*AutomationImpl. For example, the Spark Button control's delegate class is SparkButtonAutomationImpl.

The delegate classes register themselves with their associated Automation class by providing the component class and their own class as input. The AutomationManager class uses the Automation class-to-delegate class map to create a delegate instance that corresponds to a component instance in the `childAddedHandler()` method.

The delegate classes are mixins, so their `init()` method is called by the SystemManager class. The `init()` method of the delegate classes:

1 Calls the `registerDelegateClass()` method of the Automation class. This method maps the class to an automation component class; for example:

```
var className:String = getQualifiedClassName(compClass);
delegateDictionary[className] = delegateClass;
```

2 Adds event listeners for the mouse and keyboard events; for example:

```
obj.addEventListener(KeyboardEvent.KEY_UP, btnKeyUpHandler, false,
EventPriority.DEFAULT+1, true);
obj.addEventListener(MouseEvent.CLICK, clickHandler, false, EventPriority.DEFAULT+1,
true);
```

These event handlers call the AutomationManager class's `recordAutomatableEvent()` method, which in turn dispatches the `AutomationRecordEvent.RECORD` events that the automation agent listens for.

All core framework and charting classes have delegate classes already created. You are not required to create any delegate classes unless you have custom components that dispatch events that you want to automate. In this case, you must create a custom delegate class for inclusion in your application.

About the agent

The agent facilitates communication between the application built with Flex and automation tools such as QTP and Segue.

When recording, the agent class is typically responsible for implementing a persistence mechanism in the automation process. It gets information about events, user sessions, and application properties and typically writes them out to a database, log file, LocalConnection, or some other persistent storage method.

When you create an agent, you compile it and its supporting classes into a SWC file. You then add that SWC file to your application by using the `include-libraries` command-line compiler option. This compiles the agent into the application, regardless of whether you instantiate that agent in the application at compile time.

A custom agent class must be a mixin, which means that its `init()` method is called by the SystemManager class. The `init()` method of the agent:

- Defines a handler for the `RECORD` events.
- Defines the environment. The environment indicates what components and their methods, properties, and events can be recorded with the automation API.

A typical custom agent class uses an XML file that contains Flex component API information. The agent typically loads this information with a call to the `URLRequest()` constructor, as the following example shows:

```
var myXMLURL:URLRequest = new URLRequest("AutomationGenericEnv.xml");
myLoader = new URLLoader(myXMLURL);
automationManager.automationEnvironment = new CustomEnvironment(new XML(source));
```

In this example, the source is an XML file that defines the Flex metadata (or environment information). This metadata includes the events and properties of the Flex components.

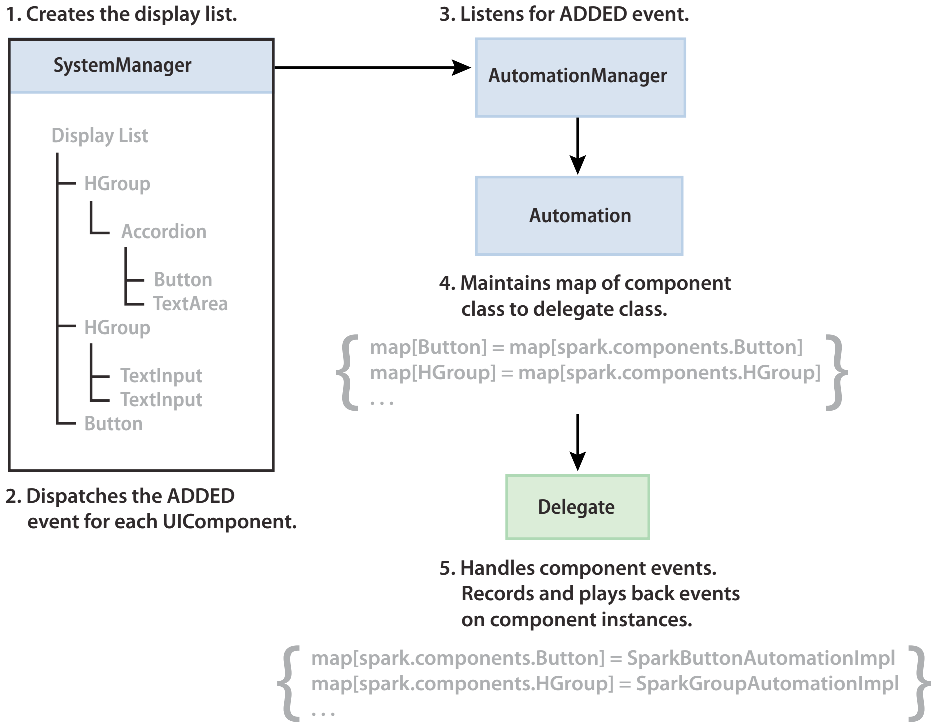
Note that representing events as XML is agent specific. The general-purpose automation API does not require it, but the XML file makes it easy to adjust the granularity of the events that are recorded.

You are not required to create an instance of your adapter in the `init()` method. You can also create this instance in the `APPLICATION_COMPLETE` event handler if your agent requires that the application must be initialized before it is instantiated.

Understanding the automation flow

When the application is initialized, the AutomationManager object is created. In its `init()` method, it adds a listener for `Event.ADDED` events.

The following image shows the order of events when the application is initialized and the AutomationManager class constructs the delegate map.


1. Creates the display list.
SystemManager
Display List
HGroup
Accordion
Button
TextArea
HGroup
TextInput
TextInput
Button
2. Dispatches the ADDED event for each UIComponent.
3. Listens for ADDED event.
AutomationManager
Automation
4. Maintains map of component class to delegate class.
map[Button] = map[spark.components.Button]
map[HGroup] = map[spark.components.HGroup]
. . .
Delegate
5. Handles component events. Records and plays back events on component instances.
map[spark.components.Button] = SparkButtonAutomationImpl
map[spark.components.HGroup] = SparkGroupAutomationImpl
. . .

1 The SystemManager class creates the display list, a tree of visible objects that make up your application.

2 Each time a new component is added, either at the root of the display list or as a child of another member of the display list, SystemManager dispatches an `Event.ADDED` event.

3 The AutomationManager listens for the `ADDED` event. In its `ADDED` event handler, it calls methods on the Automation class. It then instantiates the delegate for that class.

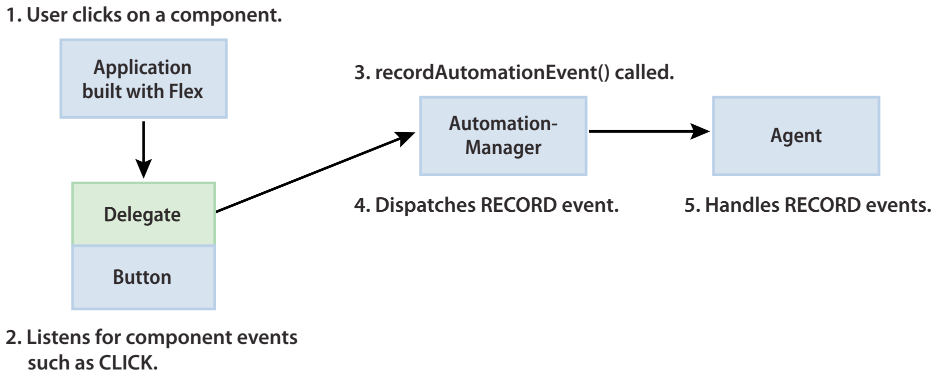
4 The Automation class maps each component in the display list to its full class name.

5 When it is created, the delegate class adds a reference to its instance in the delegate class map. The delegate class then handles events during record and play-back sequences.

The delegate is now considered *registered* with the component. It adds event listeners for the component's events and calls the AutomationManager when the component triggers those events.

After the components in the display list are instantiated and mapped to instances of their delegate classes, the AutomationManager is ready to listen for events and forward them to the agent for processing.

The following image shows the flow of operation when a user performs an action that is a recordable event. In this case, the user clicks a Button control in the application.


1. User clicks on a component.
Application built with Flex
3. recordAutomationEvent() called.
Automation-Manager
Agent
Delegate
Button
4. Dispatches RECORD event.
5. Handles RECORD events.
2. Listens for component events such as CLICK.

1 The user clicks the Button control in the application. The SystemManager dispatches a `MouseEvent.CLICK` event.

2 The SparkButtonAutomationImpl class, the Button control's automation delegate, listens for `click` events. In the delegate's `click` event handler, the delegate calls the AutomationManager `recordAutomationEvent()` method. (It is likely that the button also defines a `click` event handler to respond to the user action, but that is not shown.)

3 The AutomationManager's `recordAutomationEvent()` method dispatches an `AutomationRecordEvent.RECORD` event. In that event, the `replayableEvent` property points to the original `click` event.

4 The custom agent class listens for `RECORD` events. When it receives the `RECORD` event, it uses the `replayableEvent` property to access the properties of the original event.

5 The agent records the event properties in a database, logs the event properties, or gets information about the user before recording them.

Creating agents

You create an agent as a SWC file and link it into the application by using the `include-libraries` compiler option. You can link multiple agents in any number of SWC files to the same application. However, to use multiple agents at the same time, you must use the same environment configuration files for all agents.

The general process for creating a custom agent is:

- Mark the agent class as a mixin; this triggers a call to a static `init()` method from the SystemManager class on application start up.
- Get a reference to the AutomationManager class.
- Add event listeners for the `APPLICATION_COMPLETE` and `RECORD` events.
- Load the environment information (Flex metadata that describes the objects and operations of the application). Environment information can be an XML file or it can be in some other data format.

- Define a static `init()` method that creates an instance of the agent.
- Define a method that handles `RECORD` events. In that method, you can access:
  - Automation details such as the automation name
  - User information such as the FlexSession object (through a RemoteObject)
  - The event that triggered the `RECORD` event and its target

The `RECORD` event handler in the agent gets an `AutomationRecordEvent` whose target is the object on which recording happened. The `automationManager.createID()` method converts the object to a string that can be recorded on the screen. Some tools may require the entire automation hierarchy that needs to be generated in this method.

You also use the custom agent class to enable and disable recording of automation events.

Optimizing applications

Improving client-side performance

Tuning software to achieve maximum performance is not an easy task. You must commit to producing efficient implementations and monitor software performance continuously during the software development process.

Employ the following general guidelines when you test applications for performance, such as using the getTimer() method and checking initialization time.

Before you begin actual testing, you should understand some of the influences that client settings can have on performance testing. For more information, see “Configuring the client environment” on page 2303.

In addition to these techniques, you should also consider using the Adobe® Flex® profiler in Adobe® Flash® Builder™. For more information, see Profiling Tools in Flash Builder.

General guidelines

You can use the following general guidelines when you improve your application and the environment in which it runs:

- Set performance targets early in the software design stage. If possible, try to estimate an acceptable performance target early in the application development cycle. Certain usage scenarios dictate the performance requirements. It would be disappointing to fully implement a product feature and then find out that it is too slow to be useful.
- Understand performance characteristics of the application framework, and employ the strategies that maximize the efficiency of components and operations.
- Understand performance characteristics of the application code. In medium-sized or large-sized projects, it is common for a product feature to use codes or components written by other developers or by third-party vendors. Knowing what is slow and what is fast in dependent components and code is essential in getting the design right.
- Do not attempt to test a large application’s performance all at once. Rather, test small pieces of the application so that you can focus on the relevant results instead of being overwhelmed by data.
- Test the performance of your application early and often. It is always best to identify problem areas early and resolve them in an iterative manner, rather then trying to shove performance enhancements into existing, poorly performing code at the end of your application development cycle.

- Avoid optimizing code too early. Even though early testing can highlight performance hot spots, refrain from fixing them while you are still developing those areas of the application; doing so might unexpectedly delay the implementation schedule. Instead, document the issues and prioritize all the performance issues as soon as your team finishes the feature implementation.

Testing applications for performance

You can use various techniques to test start-up and run-time performance of your applications, such as monitoring memory consumption, timing application initialization, and timing events. The Flex profiler provides this type of information without requiring you to write any additional code. If you are using Flash Builder, you should use the profiler for testing your application's performance. For more information, see Profiling Tools in Flash Builder.

Calculating application initialization time

One approach to performance profiling is to use code to gauge the start-up time of your application. This can help identify bottlenecks in the initialization process, and reveal deficiencies in your application design, such as too many components or too much reliance on nested containers.

The getTimer() method in flash.utils returns the number of milliseconds that have elapsed since Adobe® Flash® Player or Adobe AIR™ was initialized. This indicates the amount of time since the application began playing. The Timer class provides a set of methods and properties that you can use to determine how long it takes to execute an operation.

Before each update of the screen, Flash Player calls the set of functions that are scheduled for the update. Sometimes, a function should be called in the next update to allow the rest of the code scheduled for the current update to execute. You can instruct Flash Player or AIR to call a function in the next update by using the callLater() method. This method accepts a function pointer as an argument. The method then puts the function pointer on a queue, so that the function is called the next time the player dispatches either a `render` event or an `enterFrame` event.

The following example records the time it takes the Application object to create, measure, lay out, and draw all of its children. This example does not include the time to download the SWF file to the client, or to perform any of the server-side processing, such as checking the Flash Player version, checking the SWF file cache, and so on.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- optimize/ShowInitializationTime.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="callLater(showInitTime)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import flash.utils.Timer;
        [Bindable]
        public var t:String;
        private function showInitTime():void {
            // Record the number of ms since the player was initialized.
            t = "App startup: " + getTimer() + " ms";
        }
    ]]></fx:Script>
    <s:Label id="tb1" text="{t}"/>
</s:Application>
```

This example uses the `callLater()` method to delay the recording of the startup time until after the application finishes and the first screen updates. The reason that the `showInitTime` function pointer is passed to the `callLater()` method is to make sure that the application finishes initializing itself before calling the `getTimer()` method.

For more information on using the `callLater()` method, see "Using the callLater() method" on page 2340.

Calculating elapsed time

Some operations take longer than others. Whether these operations are related to data loading, instantiation, effects, or some other factor, it's important for you to know how long each aspect of your application takes.

You can calculate elapsed time from application startup by using the getTimer() method. The following example calculates the elapsed times for the `preinitialize` and `creationComplete` events for all the form elements. You can modify this example to show individual times for the initialization and creation of each form element.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- optimize/ShowElapsedTime.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="init()"
    height="750">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var dp:ArrayCollection = new ArrayCollection ([
            {food:"apple", type:"fruit", color:"red"},
            {food:"potato", type:"vegetable", color:"brown"},
            {food:"pear", type:"fruit", color:"green"},
            {food:"orange", type:"fruit", color:"orange"},
            {food:"spinach", type:"vegetable", color:"green"},
            {food:"beet", type:"vegetable", color:"red"}
        ]);
        public var sTime:Number;
        public var eTime:Number;
        public var pTime:Number;
        private function init():void {
            f1.addEventListener("preinitialize", logPreInitTime, true);
            f1.addEventListener("creationComplete", logCreationCompTime, true);
        }
        private var isFirst:Boolean = true;
        private function logPreInitTime(e:Event):void {
            // Get the time when the preinitialize event is dispatched.
            sTime = getTimer();

           trace("Preinitialize time for " + e.target + ": " + sTime.toString());
        }
        private function logCreationCompTime(e:Event):void {
            // Get the time when the creationComplete event is dispatched.
            eTime = getTimer();

            /* Use target rather than currentTarget because these events are
```

```
               triggered by each child of the Form control during the capture
               phase. */
            trace("CreationComplete time for " + e.target + ": " + eTime.toString());
        }

    ]]></fx:Script>
    <s:Form id="f1">
        <s:FormHeading label="Sample Form" id="fh1"/>
        <s:FormItem label="List Control" id="fi1">
            <s:List dataProvider="{dp}" labelField="food" id="list1"/>
        </s:FormItem>
        <s:FormItem label="DataGrid control" id="fi2">
            <s:DataGrid width="200" dataProvider="{dp}" id="dg1"/>
        </s:FormItem>
        <s:FormItem label="Date controls" id="fi3">
            <mx:DateChooser id="dc"/>
            <mx:DateField id="df"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Calculating memory usage

You use the `totalMemory` property in the System class to find out how much memory has been allocated to Flash Player or AIR on the client. The `totalMemory` property represents all the memory allocated to Flash Player or AIR, not necessarily the memory being used by objects. Depending on the operating system, Flash Player or AIR will be allocated more or less resources and will allocate memory with what is provided.

You can record the value of `totalMemory` over time by using a Timer class to set up a recurring interval for the timer event, and then listening for that event.

The following example displays the total amount of memory allocated (totmem) to Flash Player at 1-second intervals. This value will increase and decrease. In addition, this example shows the maximum amount of memory that had been allocated (maxmem) since the application started. This value will only increase.

```
<?xml version="1.0"?>
<!-- optimize/ShowTotalMemory.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initTimer()">
    <fx:Script><![CDATA[
        import flash.utils.Timer;
        import flash.events.TimerEvent;
        [Bindable]
        public var time:Number = 0;
        [Bindable]
        public var totmem:Number = 0;
        [Bindable]
        public var maxmem:Number = 0;
        public function initTimer():void {
            // The first parameter is the interval (in milliseconds). The
            // second parameter is number of times to run (0 means infinity).
            var myTimer:Timer = new Timer(1000, 0);
            myTimer.addEventListener("timer", timerHandler);
            myTimer.start();
        }

        public function timerHandler(event:TimerEvent):void {
            time = getTimer()
            totmem = flash.system.System.totalMemory;
            maxmem = Math.max(maxmem, totmem);
        }
    ]]></fx:Script>

    <s:Form>
        <s:FormItem label="Time:">
            <s:Label text="{time} ms"/>
        </s:FormItem>
        <s:FormItem label="totalMemory:">
            <s:Label text="{totmem} bytes"/>
        </s:FormItem>
        <s:FormItem label="Max. Memory:">
            <s:Label text="{maxmem} bytes"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

Configuring the client environment

When testing applications for performance, it is important to configure the client properly.

Choosing the version of Flash Player

When you test your applications for performance, use the standard version of Adobe® Flash® Player or AIR rather than the debugger version of Flash Player or ADL, if possible. The debugger version of Flash Player provides support for the trace() method and the Logging API. Using logging or the `trace()` method can significantly slow player performance, because the player must write log entries to disk while running the application.

If you do use the debugger version of Flash Player, you can disable logging and the `trace()` method by setting the `TraceOutputFileEnable` property to 0 in your mm.cfg file. You can also set the `omit-trace-statements` compiler option to `false`.

You can keep `trace()` logging working, but disable the Logging API that you might be using in your application, by setting the logging level of the TraceTarget logging target to `NONE`, as the following example shows:

```
myLogger.log(LogEventLevel.NONE, s);
```

For performance testing, consider writing run-time test results to text components in the application rather than calling the `trace()` method so that you can use the standard version of Flash Player and not the debugger version of Flash Player.

For more information about configuring `trace()` method output and logging, see “Logging” on page 2222.

Disabling SpeedStep

If you are running performance tests on a Windows laptop computer, disable Intel SpeedStep functionality. SpeedStep toggles the speed of the CPU to maximize battery life. SpeedStep can toggle the CPU at unpredictable times, which makes the results of a performance test less accurate than they would otherwise be.

1 Select Start > Settings > Control Panel.

2 Double-click the Power Settings icon. The Power Options Properties dialog box displays.

3 Select the Power Schemes tab.

4 Select High System Performance from the Power Schemes drop-down box.

5 Click OK.

Changing timeout length

When you test your application, be aware of the `scriptTimeLimit` property. If an application takes too long to initialize, Flash Player warns users that a script is causing Flash Player to run slowly and prompts the user to abort the application. If this is the situation, you can set the `scriptTimeLimit` property of the `<s:Application>` tag to a longer time so that the application has enough time to initialize.

However, the default value of the `scriptTimeLimit` property is 60 seconds, which is also the maximum, so you can only increase the value if you have previously set it to a lower value. You rarely need to change this value.

The following example sets the `scriptTimeLimit` property to 30:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- optimize/ChangeScriptTimeLimit.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    scriptTimeLimit="30">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
</s:Application>
```

Preventing client-side caching

When you test performance, ensure that you are not serving files from the local cache to Flash Player. Otherwise, this can give false results about download times. Also, during development and testing, you might want to change aspects of the application such as embedded images, but the browser continues to use the old images from your cache.

If the date and time in the If-Modified-Since request header matches the date and time in the Last-Modified response header, the browser loads the SWF file from its cache. Then the server returns the “304 Not Modified” message. If the Last-Modified header is more recent, the server returns the SWF file.

You can use the following techniques to disable client-side caching:

- Delete the Flex files from the browser’s cache after each interaction with your application. Browsers typically store the SWF file and other remote assets in their cache. On Microsoft Internet Explorer in Windows XP, for example, you can delete all the files in c:\Documents and Settings\username\Local Settings\Temporary Internet Files to force a refresh of the files on the next request. For more information, see “Caching” on page 122.
- Set the HTTP headers for the SWF file request in the HTML wrapper to prevent caching of the SWF file on the client. The following example shows how to set headers that prevent caching in JSP:

  ```
  // Set Cache-Control to no-cache.
  response.setHeader("Cache-Control", "no-cache");
  // Prevent proxy caching.
  response.setHeader("Pragma", "no-cache");
  // Set expiration date to a date in the past.
  response.setDateHeader("Expires", 946080000000L); //Approx Jan 1, 2000
  // Force always modified.
  response.header("Last-Modified", new Date());
  ```

  Note that in some cases, setting the `Pragma` header to `"no-cache"` can cause a runtime error with GET requests over SSL with the HTTPService class. In this case, setting just the `Cache-control` header should work.

Marc Speck has a blog entry that describes the issue and presents several solutions.

Reducing SWF file sizes

You can improve initial user experience by reducing the time it takes to start an application. Part of this time is determined by the download process, where the SWF file is returned from the server to the client. The smaller the SWF file, the shorter the download wait. In addition, reducing the size of the SWF file also results in a shorter application initialization time. Larger SWF files take longer to unpack in Flash Player.

The mxmlc compiler includes several options that can help reduce SWF file size.

Using the bytecode optimizer

The bytecode optimizer can reduce the size of the application's SWF file by using bytecode merging and peephole optimization. Peephole optimization removes redundant instructions from the bytecode.

If you are using Flash Builder or the mxmlc command-line compiler, you can set the `optimize` compiler option to `true`, as the following example shows:

```
mxmlc -optimize=true MyApp.mxml
```

The default value of the `optimize` option is `true`.

Disabling debugging

Disabling debugging can make your SWF files smaller. When debugging is enabled, the Flex compilers include line numbers and other navigational information in the SWF file that are only used in a debugging environment. Disabling debugging reduces functionality of the fdb command-line debugger and the debugger built into Flash Builder.

To disable debugging in Flash Builder, select Project > Export Release Build when you build your application.

To disable debugging when using the command-line compiler, set the `debug` compiler option to `false`. The default value for the mxmlc compiler is `false`. The default value for the compc compiler is `true`. Setting this to `false` reduces the size of hte SWF file inside the SWC file.

For more information about debugging, see "Command-line debugger" on page 2209.

Using the size report

The `size-report` compiler option outputs a high-level summary of the size of each type of data within your application's SWF file. You can use this information to identify problem areas in an application. For example, if you embed multiple fonts, you can compare the size that each embedded font takes up. If one font is considerably larger than the others, you could reduce the number of symbols embedded by that font.

The following example creates a size report called mysizereport.xml for the MainApp application:

```
mxmlc -size-report=mysizereport.xml MainApp.mxml
```

To use this option in Flash Builder, you must add it to the Additional Compiler Arguments field.

The following table describes the areas of the report:

Report tag	**Description**
`<swf>`	The compressed and uncompressed SWF file size, in bytes.
`<headerData>`	The size of the actual SWF file format header, product info, and other marker data.
`<actionScript>`	The ActionScript data associated with this SWF file. For non-debug SWFs, ActionScript bytecode (abc) and constant pool data is consolidated into a single block within the frame that it is associated with. For debug SWF or SWC files, the ActionScript bytecode blocks are generally broken down by symbol. All entries are sorted largest to smallest.
`<frames>`	A frame by frame summary of all script and tag data for each frame. Applications usually consist of two or more frames, with the first frame containing all preloader logic as well as the top level SystemManager definition.
`<frameData>`	Additional SWF tags that are related to frame definitions. For example, frame labels, symbol definitions, export definitions, and showFrame tags.
`<bitmaps>`	The embedded bitmap data. All entries are sorted largest to smallest. If appropriate, the symbol name of each bitmap is provided. The original file name of the embedded asset is not provided.
`<fonts>`	The embedded font data. All entries are sorted largest to smallest. If appropriate, the symbol name of each font is provided. The original font face name or path is not provided, nor any enumeration of the embedded glyphs.
`<sprites>`	The sprite data. All entries are sorted largest to smallest. If appropriate, the symbol name of each sprite definition is provided.
`<shapes>`	The shape data. All entries are sorted largest to smallest. If appropriate, the symbol name of each shape definition is provided.
`<bindaryData>`	The generic binary data. This includes embedded SWF files, PixelBender shaders, or any other miscellaneous embed data. All entries are sorted largest to smallest. If appropriate, the symbol name of each data definition is provided.The original file name for each asset is not provided.
`<sounds>`	The embedded sound data. All entries are sorted largest to smallest. If appropriate, the symbol name of each sound is provided.The original file name for each asset is not provided.
`<videos>`	The embedded video data. All entries are sorted largest to smallest. If appropriate, the symbol name of each video asset is provided.The original file name for each asset is not provided.

The following is a sample size report:

```
<?xml version="1.0" encoding="UTF-8"?>
<report>
  <swf size="614011" compressedSize="318895">
    <!-- Header data (SWF attributes, product info, markers, etc.) -->
    <headerData totalSize="533">
      <data type="metaData" size="465"/>
      <data type="productInfo" size="28"/>
      <data type="swfHeader" size="21"/>
      <data type="fileAttributes" size="6"/>
      <data type="scriptLimits" size="6"/>
      <data type="backgroundColor" size="5"/>
      <data type="endMarker" size="2"/>
    </headerData>
    <!-- Cumulative frame size summary. -->
    <frames totalSize="613478">
      <frame name="_MultipleFaces_mx_managers_SystemManager" size="76648" frame="1"/>
      <frame name="MultipleFaces" size="536830" frame="2"/>
    </frames>
    <!-- Actionscript code and constant data. -->
    <actionScript totalSize="501609">
      <abc name="frame2" size="425053" frame="2"/>
      <abc name="frame1" size="76556" frame="1"/>
    </actionScript>
    <!-- defineFont/2/3/4. -->
    <fonts totalSize="97660">
      <font name="MultipleFaces__embed__font_myFont_medium_italic_1704731415" 
fontName="myFont" size="34180" frame="2"/>
      <font name="MultipleFaces__embed__font_myFont_bold_normal_673776644" fontName="myFont" 
size="33472" frame="2"/>
      <font name="MultipleFaces__embed__font_myFont_medium_normal_1681983489" 
fontName="myFont" size="30008" frame="2"/>
    </fonts>
    <!-- defineSprite. -->
    <sprites totalSize="19">
      <sprite 
name="_MultipleFaces_Styles__embed_css_Assets_swf_mx_skins_cursor_BusyCursor_2036984981" 
size="19" frame="2"/>
    </sprites>
    <!-- defineShape/2/3/4. -->
    <shapes totalSize="261">
      <shape size="261" frame="2"/>
    </shapes>
    <!-- SWF, Pixel Bender, or other miscellaneous embed data. -->
    <binaryData totalSize="12506">
      <data name="mx.graphics.shaderClasses.SaturationShader_ShaderClass" size="2298" 
frame="2"/>
      <data name="mx.graphics.shaderClasses.HueShader_ShaderClass" size="2268" frame="2"/>
      <data name="mx.graphics.shaderClasses.SoftLightShader_ShaderClass" size="1920" 
frame="2"/>
      <data name="mx.graphics.shaderClasses.LuminosityShader_ShaderClass" size="1282" 
frame="2"/>
      <data name="mx.graphics.shaderClasses.ColorShader_ShaderClass" size="1272" frame="2"/>
      <data name="mx.graphics.shaderClasses.ColorBurnShader_ShaderClass" size="1144" 
```

```
frame="2"/>
      <data name="mx.graphics.shaderClasses.ColorDodgeShader_ShaderClass" size="1074"
frame="2"/>
      <data name="mx.graphics.shaderClasses.ExclusionShader_ShaderClass" size="624"
frame="2"/>
      <data name="mx.graphics.shaderClasses.LuminosityMaskShader_ShaderClass" size="624"
frame="2"/>
    </binaryData>
    <!-- Additional frame tags (symbolClass, exportAssets, showFrame, etc). -->
    <frameData totalSize="1423">
      <tag type="symbolClass" size="774" frame="2"/>
      <tag type="exportAssets" size="539" frame="2"/>
      <tag type="symbolClass" size="47" frame="1"/>
      <tag type="frameLabel" size="43" frame="1"/>
      <tag type="frameLabel" size="16" frame="2"/>
      <tag type="showFrame" size="2" frame="1"/>
      <tag type="showFrame" size="2" frame="2"/>
    </frameData>
  </swf>
</report>
```

Using strict mode

When you set the `strict` compiler option to `true`, the compiler verifies that definitions and package names in `import` statements are used in the application. If the imported classes are not used, the compiler reports an error.

The following example shows some examples of when strict mode throws a compiler error:

```
package {
    import flash.utils.Timer; // Error. This class is not used.
    import flash.printing.* // Error. This class is not used.
    import mx.controls.Button; // Error. This class is not used.
    import mx.core.Application; // No error. This class is used.

    public class Foo extends Application {
    }
}
```

The `strict` option also performs compile-time type checking, which provides a small optimization increase in the application at run time.

The default value of the `strict` compiler option is `true`.

Examining linker dependencies

To find ways to reduce SWF file sizes, you can look at the list of ActionScript classes that are linked into your SWF file.

You can generate a report of linker dependencies by using the `link-report` compiler option. This option takes a single file name. The compiler generates a report, and writes it to the specified file, that shows an application’s linker dependencies in an XML format.

The following example shows the dependencies for the ProgrammaticSkin script as it appears in the linker report:

```
<script name="C:\flex3sdk\frameworks\libs\framework.swc(mx/skins/ProgrammaticSkin)"
mod="1141055632000" size="5807">
    <def id="mx.skins:ProgrammaticSkin"/>
    <pre id="mx.core:IFlexDisplayObject"/>
    <pre id="mx.styles:IStyleable"/>
    <pre id="mx.managers:ILayoutClient"/>
    <pre id="flash.display:Shape"/>
    <dep id="String"/>
    <dep id="flash.geom:Matrix"/>
    <dep id="mx.core:mx_internal"/>
    <dep id="uint"/>
    <dep id="mx.core:UIComponent"/>
    <dep id="int"/>
    <dep id="Math"/>
    <dep id="Object"/>
    <dep id="Array"/>
    <dep id="mx.core:IStyleClient"/>
    <dep id="Boolean"/>
    <dep id="Number"/>
    <dep id="flash.display:Graphics"/>
</script>
```

The following table describes the tags used in this file:

Tag	Description
`<script>`	Indicates the name of a compilation unit used in the creation of the application SWF file. Compilation units must contain at least one public definition, such as a class, function, or namespace. The `name` attribute shows the origin of the script, either from a source file or from a SWC file (for example, frameworks.swc). If you set the value of the `keep-generated-actionscript` compiler argument to `true`, all classes in the generated folder are listed as scripts in this file. The `size` attribute shows the class' uncompressed size, in bytes. The `mod` attribute shows the time stamp when the script was created.
`<def>`	Indicates the name of a definition. A definition, like a script, can be a class, function, or namespace.
`<pre>`	Indicates a definition that must be linked in to the SWF file before the current definition is linked in. This tag means prerequisite. For class definitions, this tag shows the direct parent class (for example, flash.events:Event), plus all implemented interfaces (for example, mx.core:IFlexDisplayObject and mx.managers:ILayoutClient) of the class.
`<dep>`	Indicates other definitions that this definition depends on (for example, String, _ScrollBarStyle, and mx.core:IChildList). This is a reference to a definition that the current script requires. Some script definitions have no dependencies, so the `<script>` tag might have no `<dep>` child tags.
`<ext>`	Indicates a dependency to an asset that was not linked in. These dependencies show up in the linker report when you use the `external-library-path`, `externs`, or `load-externs` compiler options to add assets to the SWF file.

You can examine the list of prerequisites and dependencies for your application definition. You do this by searching for your application's root MXML file by its name; for example, MyApp.mxml. You might discover that you are linking in some classes inadvertently. When writing code, it is common to make a reference to a class but not actually require that class in your application. That reference causes the referenced class to be linked in, and it also links in all the classes on which the referenced class depends.

If you look through the linker report, you might find that you are linking in a class that is not needed. If you do find an unneeded class, try to identify the linker dependency that is causing the class to be linked in, and try to find a way to rewrite the code to eliminate that dependency.

Viewing SWC file dependencies

The swcdepends command line tool has a small set of options that lets you view dependencies on SWC files in your application. This tool is located in the sdk/bin directory. By default, the tool outputs a list of SWC files ordered by dependencies. For each SWC file in the list, the tool shows a sub-list of SWC files that the SWC file is dependent on.

The output is sorted by the number of dependencies. SWC files with the fewest dependencies are listed first, followed by those with the most dependencies. For each SWC file in the list, SWC files that are dependent on it are listed below it and indented in the output.

The swcdepends tool supports all the options of the mxmlc command line compiler. In addition, it has several options that let you customize the output. The following table describes these options:

Option	Description
`dependency.minimize-dependency-set=true|false`	Determines whether all SWC files are included for depencies, or just the first one. For example, some classes are defined in more than one SWC file. When this option is `true`, only the first SWC file in which the class is defined is added as a dependency. When this option is `false`, all SWC files that define a class appear as dependencies.
`dependency.show-external-classes=false|true`	Shows classes that cause a dependency from one SWC to another. The default value is `false`.
`dependency.show-swcsswc_filename`	Limits the output to show just the given SWC files. Does not change the set of SWC files used to determine dependencies.
`dependency.show-types`	Shows the dependency type(s) of a class.
`dependency.typestype`	Limits the dependency checking to only the specified types. By default all the dependency types are specified. This option is useful when you want to see what RSLs are needed by the RSL from a given SWC file. The set of types is as follows: • e - expression • i - inheritance • n - namespace • s - signature You can specify "i" to see all the SWC files that have an inheritance dependency on a SWC file. Some or all of the dependent SWC files must be RSLs; as a result, be sure that the required classes are loaded before the RSL for the SWC file is loaded. You can specify multiple dependency types by using a comma-separated list, as the following example shows: `-dependency.types=i,e` This example limits the dependency list to external classes with either inheritance or expression dependency types.

Avoiding initializing unused classes

Some common ways to avoid unnecessary references include avoiding initializing classes that you do not use and performing type-checking with the `getQualifiedClassName()` method.

The following example checks if the class is a Button control. This example forces the compiler to include a Button in the SWF file, even if the child is not a Button control and the entire application has no Button controls.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- optimize/UnusedClasses.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="checkChildType()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        import spark.components.Button;
        [Bindable]
        private var s:String;
        public function checkChildType():void {
            var child:DisplayObject = getChildAt(0);
            var childIsButton:Boolean = child is spark.components.Button;
            s = "child is spark.components.Button: " + childIsButton.toString(); // False.
        }
    ]]></fx:Script>
    <!-- This control is here so that the getChildAt() method succeeds. -->
    <s:DataGrid/>
    <s:Group>
        <s:Label text="{s}"/>
    </s:Group>
</s:Application>
```

You can use the `getQualifiedClassName()` method to accomplish the same task as the previous example. This method returns a String that you can compare to the name of a class without causing that class to be linked into the SWF.

The following example does not create a linker dependency on the Button control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- optimize/GetQualifiedClassNameExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="checkChildType()">
    <s:layout>
         <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
    [Bindable]
    private var s:String;
    [Bindable]
    private var t:String;
    public function checkChildType():void {
        var child:DisplayObject = vg1.getChildAt(0);
        var childClassName:String = getQualifiedClassName(child);
        var childIsButton:Boolean = childClassName == "spark.components::Button"
        s = "child class name = Button (" + childIsButton + ")";
        t = "child is " + childClassName;
    }
    ]]></fx:Script>
    <s:VGroup id="vg1">
        <s:DataGrid/>
        <s:Label text="{s}"/>
        <s:Label text="{t}"/>
    </s:VGroup>
</s:Application>
```

Externalizing assets

There are various methods of externalizing assets used by your applications; these include:

- Using modules
- Using run-time stylesheets
- Using Runtime Shared Libraries (RSLs)
- Loading assets at run time rather than embedding them

This section describes loading assets at run time. For information about modules and run-time stylesheets, see "Modular applications" on page 138 and "Loading style sheets at run time" on page 1547. For information about RSLs, see "Using RSLs to reduce SWF file size" on page 2315.

One method of reducing the SWF file size is to externalize assets; that is, to load the assets at run time rather than embed them at compile time. You can do this with assets such as images, SWF files, and sound files.

Embedded assets load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded assets also require you to recompile your applications whenever your asset changes.

The following example embeds the butterfly.gif file into the application at compile time:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- optimize/EmbedAtCompileTime.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
         <s:VerticalLayout/>
    </s:layout>
    <s:Image source="@Embed(source='../assets/butterfly.gif')"/>
</s:Application>
```

The following example loads the butterfly.gif file into the application at run time:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- optimize/EmbedAtRunTime.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
         <s:VerticalLayout/>
    </s:layout>
    <s:Image source="assets/butterfly.gif"/>
</s:Application>
```

The only supported image type that you cannot load at run time is SVG. Flash Player and AIR require that the compiler transcodes that file type at compile time. The Player and AIR runtime cannot transcode that file type at run time.

When you load SWF files from domains that are not the same as the loading SWF file, you must use a crossdomain.xml file or other mechanism to enable the proper permissions. For more information on using the crossdomain.xml file, see “Using cross-domain policy files” on page 125.

Using character ranges for embedded fonts

By specifying a range of symbols that compose the face of an embedded font, you reduce the size of an embedded font. Each character in a font must be described; if you remove some of these characters, it reduces the overall size of the description information that Flex must include for each embedded font.

You can set the range of glyphs in the flex-config.xml file or in the `@font-face` declaration in each MXML file. You specify individual characters or ranges of characters using the Unicode values for the characters, and you can set multiple ranges for each font declaration.

In CSS, you can set the Unicode range with the `unicodeRange` property, as the following example shows:

```
@font-face {
    src:url("../assets/MyriadWebPro.ttf");
    fontFamily: myFontFamily;
    unicodeRange:
        U+0041-005A, /* Upper-Case [A..Z] */
        U+0061-007A, /* Lower-Case a-z */
        U+0030-0039, /* Numbers [0..9] */
        U+002E-002E; /* Period [.] */
    embedAsCFF:true;
}
```

In the flex-config.xml file, you can set the Unicode range with the `<language-range>` block, as the following example shows:

```
<language-range>
    <lang>Latin I</lang>
    <range>U+0020,U+00A1-00FF,U+2000-206F,U+20A0-20CF,U+2100-2183</range>
</language-range>
```

For more information, see “Fonts” on page 1568.

Using multiple SWF files

One way to reduce the size of an application’s file is to break the application up into logical parts that can be sent to the client and loaded over a series of requests rather than all at once. By breaking a monolithic application into smaller applications, users can interact with your application more quickly, but possibly experience some delays while the application is running.

One approach is to use the SWFLoader control to load sub-applications. This technique can work with SWF files that add graphics or animations to an application, or SWF files that act as stand-alone applications inside the main application. It also provides some level of interoperability between the main application and loaded sub-applications. If you load SWF files that require a large amount of user interaction, however, consider building them as custom components.

When loading sub-applications into a main application, you should be aware of the following factors:

Versioning SWF files produced with earlier versions of Flex or ActionScript may not work properly when loaded with the SWFLoader control. You can use the `loadForCompatibility` property of the SWFLoader control to ensure that sub-applications loaded into a main application will work, even if the applications were compiled with a different version of the compiler.

Security When loading sub-applications, especially ones that were created by a third-party, you should consider loading them into their own SecurityDomain. While this places additional limitations on the level of interoperability between the main application and the sub-application, it ensures that the content is safe from attack.

For more information about creating and loading sub-applications, see “Developing and loading sub-applications” on page 176.

Rather than loading SWF files into the main application with the SWFLoader control, consider having the SWF files communicate with each other as separate applications. You can do this with local SharedObjects, LocalConnection objects, or with the ExternalInterface API.

You can also use modules to externalize sections of your application into dynamically loaded SWFs. While modules do not work as standalone applications, they can be loaded and unloaded in your main application. For more information, see “Modular applications” on page 138.

Comparing dynamic and static linking

Most large applications use libraries of ActionScript classes and components. You must decide whether to use static or dynamic linking when using these libraries in your applications.

When you use *static linking*, the compiler includes all components, classes, and their dependencies in the application SWF file when you compile the application. The result is a larger SWF file that takes longer to download but loads and runs quickly because all the code is in the SWF file. To compile your application that uses libraries and to statically link those definitions into your application, you use the `library-path` and `include-libraries` options to specify the locations of SWC files.

Dynamic linking is when some classes used by an application are left in an external file that is loaded at run time. The result is a smaller SWF file size for the main application, but the application relies on external files that are loaded during run time.

To dynamically link classes and components, you compile a library. You then instruct the compiler to exclude that library's contents from the application SWF file. You must still provide link-checking at compile time even though the classes are not going to be included in the final SWF file.

You use dynamic linking by creating component libraries and compiling them with your application by using the `external-library-path`, `externs`, or `load-externs` compiler options. These options instruct the compiler to exclude resources defined by their arguments from inclusion in the application, but to check links against them and prepare to load them at run time. The `external-library-path` option specifies SWC files or directories for dynamic linking. The `externs` option specifies individual classes or symbols for dynamic linking. The `load-externs` option specifies an XML file that describes which classes to use for dynamic linking. This XML file has the same syntax as the file produced by the `link-report` compiler option.

For more information about linking, see "About linking" on page 254. For more information about compiler options, see "Flex compilers" on page 2164.

Using RSLs to reduce SWF file size

One way to reduce the size of your application's SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but must be transferred only once to the client. These shared files are known as *Runtime Shared Libraries* (RSLs).

If you have multiple applications but those applications share a core set of custom components or classes, your users will be required to download those assets only once as an RSL. The applications that share the assets in the RSL use the same cached RSL as the source for the libraries as long as they are in the same domain. The resulting file size for your applications can be reduced. The benefits increase as the number of applications that use the RSL increases.

When you create an RSL, be sure to optimize it prior to deployment. This removes debugging information as well as unnecessary metadata from the RSL, which can dramatically reduce its size.

By default, Flex compiles your applications against the framework RSLs.

For more information, see "Runtime Shared Libraries" on page 253.

Application coding

The MXML language provides a rich set of controls and classes that you can use to create interactive applications. This richness sometimes can reduce performance. However, there are some techniques that a Flex developer can use to improve the run-time performance of the application.

To measure the effects of the following techniques, you should use the Flex profiler. For more information, see Profiling Tools in Flash Builder.

Object creation and destruction

Object creation is the task of instantiating all the objects in your application. These objects include controls, components, and objects that contain data and other dynamic information. Optimizing the process of object creation and destruction can result in significant performance gains. *Object destruction* is the act of reallocating memory for objects after all references to those objects have been removed. This task is carried out by the garbage collector at regular intervals. You can improve the frequency that Flash Player and AIR destroy objects by removing references to objects.

No single task during application initialization takes up the most time. The best way to improve performance is to create fewer objects. You can do this by deferring the instantiation of objects, or changing the order in which they are created to improve perceived performance.

Using deferred creation

To improve the start-up time of your application, you can minimize the number of objects that are created when the application is first loaded. If a user-interface component is not initially visible at start up, create that component only when you need it. This is called deferred creation. Containers that have multiple views, such as an Accordion, provide built-in support for this behavior. You can use ActionScript to customize the creation order of multiple-view containers or defer the creation of other containers and controls.

To use deferred creation, you set the value of a component's creationPolicy property to `all`, `auto`, or `none`. If you set it to `none`, Flex does not instantiate a control's children immediately, but waits until you instruct Flex to do so. If you set the `creationPolicy` property on a container, then the children of that container are affected by its setting. So, for example, if you set the `creationPolicy` property to none on a Panel container, then all children of the Panel are not created when the application starts up. The children are only created when you manually instantiate them.

For Spark containers that extend SkinnableContainer, you call the `createDeferredContent()` method to create deferred components. Only SkinnableContainer and containers that extend it support deferred instantiation; Groups do not. When a Group is created, its children are always created.

In the following example with a Spark container, the children of the Panel container are not instantiated when the application is first loaded, but only after the user clicks the button:

```
<?xml version="1.0"?>
<!-- optimize/CreationPolicyNoneSpark.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function createButtons(e:Event):void {
            myPanel.createDeferredContent();
        }
    ]]></fx:Script>
    <s:Panel id="myPanel" title="Panel with Deferred Content" creationPolicy="none">
        <s:VGroup>
            <s:Button id="b1" label="Hurley"/>
            <s:Button id="b2" label="Jack"/>
            <s:Button id="b3" label="Sawyer"/>
        </s:VGroup>
    </s:Panel>

    <s:Button id="myButton" click="createButtons(event)" label="Create Buttons"/>
</s:Application>
```

For MX containers, call methods such as createComponentFromDescriptor() and createComponentsFromDescriptor() on the container to instantiate its children at run time.

In the following example with an MX container, the children of the VBox container are not be instantiated when the application is first loaded, but only after the user clicks the button:

```
<?xml version="1.0"?>
<!-- optimize/CreationPolicyNone.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function createButtons(e:Event):void {
            myVBox.createComponentsFromDescriptors();
        }
    ]]></fx:Script>
    <s:Panel title="VBox with Deferred Content">
        <mx:VBox id="myVBox" height="100" width="125" creationPolicy="none">
            <s:Button id="b1" label="Hurley"/>
            <s:Button id="b2" label="Jack"/>
            <s:Button id="b3" label="Sawyer"/>
        </mx:VBox>
    </s:Panel>
    <s:Button id="myButton" click="createButtons(event)" label="Create Buttons"/>

</s:Application>
```

For more information on using deferred instantiation, see “Using deferred creation” on page 2335.

Destroying unused objects

Flash Player provides built-in garbage collection that frees up memory by destroying objects that are no longer used. To ensure that the garbage collector destroys your unused objects, remove all references to that object, including the parent’s reference to the child.

For more information about garbage collection, see Profiling Tools in Flash Builder.

For Spark containers, you can call the `removeElement()`, `removeAllElements()`, or `removeElementAt()` methods to remove references to child controls. For MX containers, you can call the removeChild() or removeChildAt() method to remove references to child controls that are no longer needed.

The following example removes references to button instances from the MX myVBox and Spark myGroup containers:

```
<?xml version="1.0"?>
<!-- optimize/DestroyObjects.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function destroyButtons(e:Event):void {
            myVBox.removeChild(b1);
            myVBox.removeChild(b2);
            myVBox.removeChild(b3);

            myGroup.removeElement(b4);
        }
    ]]></fx:Script>
    <s:Panel title="VBox and Group">
        <s:VGroup>
            <mx:VBox id="myVBox" height="100" width="125">
                <s:Button id="b1" label="Hurley"/>
                <s:Button id="b2" label="Jack"/>
                <s:Button id="b3" label="Sawyer"/>
            </mx:VBox>
            <s:Group id="myGroup">
                <s:Button id="b4" label="Other"/>
            </s:Group>
        </s:VGroup>
    </s:Panel>

    <s:Button id="myButton2" click="destroyButtons(event)" label="Destroy Buttons"/>
</s:Application>
```

You can clear references to unused variables by setting them to `null` in your ActionScript; for example:

```
myDataProvider = null
```

To ensure that destroyed objects are garbage collected, you must also remove event listeners on them by using the removeEventListener() method, as the following example shows:

```
<?xml version="1.0"?>
<!-- optimize/RemoveListeners.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function initApp(e:Event):void {
            b1.addEventListener("click",myClickHandler);
            b2.addEventListener("click",myClickHandler);
            b3.addEventListener("click",myClickHandler);
        }
        private function destroyButtons(e:Event):void {
            b1.removeEventListener("click",myClickHandler);
            b2.removeEventListener("click",myClickHandler);
            b3.removeEventListener("click",myClickHandler);
            myVGroup.removeAllElements();
        }

        private function myClickHandler(e:Event):void {
            // Do something here.
        }
    ]]></fx:Script>
    <s:Panel title="VGroup with child controls">
        <s:VGroup id="myVGroup" height="100" width="125">
            <s:Button id="b1" label="Hurley"/>
            <s:Button id="b2" label="Jack"/>
            <s:Button id="b3" label="Sawyer"/>
        </s:VGroup>
    </s:Panel>

    <s:Button id="myButton" click="destroyButtons(event)" label="Destroy Buttons"/>
</s:Application>
```

You cannot call the `removeEventListener()` method on an event handler that you added inline. In the following example, you cannot call `removeEventListener()` on b1's `click` event handler, but you can call it on b2's and b3's event handlers:

```
<?xml version="1.0"?>
<!-- optimize/RemoveSomeListeners.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function initApp(e:Event):void {
            b2.addEventListener("click",myClickHandler);
            b3.addEventListener("click",myClickHandler);
        }
        private function destroyButtons(e:Event):void {
            b2.removeEventListener("click",myClickHandler);
            b3.removeEventListener("click",myClickHandler);
            myVGroup.removeAllElements();
        }

        private function myClickHandler(e:Event):void {
            // Do something here.
        }
    ]]></fx:Script>
    <s:Panel title="VGroup with child controls">
        <s:VGroup id="myVGroup" height="100" width="125">
            <s:Button id="b1" label="Hurley" click="myClickHandler(event)"/>
            <s:Button id="b2" label="Jack"/>
            <s:Button id="b3" label="Sawyer"/>
        </s:VGroup>
    </s:Panel>

    <s:Button id="myButton" click="destroyButtons(event)" label="Destroy Buttons"/>
</s:Application>
```

The *weakRef* parameter to the `addEventListener()` method provides you with some control over memory resources for listeners. A strong reference (when `weakRef` is `false`) prevents the listener from being garbage collected. A weak reference (when `weakRef` is `true`) does not. The default value of the *weakRef* parameter is `false`.

For more information about the `removeEventListener()` method, see “Events” on page 54.

Using styles

You use styles to define the look and feel of your applications. You can use them to change the appearance of a single component, or apply them globally. Be aware that some methods of applying styles are more computationally expensive than others. You can increase your application’s performance by changing the way you apply styles.

For more information about using styles, see “Styles and themes” on page 1492.

Loading stylesheets at run time

You can load stylesheets at run time by using the StyleManager. These style sheets take the form of SWF files that are dynamically loaded while your application runs.

By loading style sheets at run time, you can load images (for graphical skins), fonts, type and class selectors, and programmatic skins into your application without embedding them at compile time. This lets skins and fonts be partitioned into separate SWF files, away from the main application. As a result, the application's SWF file size is smaller, which reduces the initial download time. However, the first time a run-time style sheet is used, it takes longer for the styles and skins to be applied because Flex must download the necessary CSS-based SWF file.

Reducing calls to the setStyle() method

Run-time cascading styles are very powerful, but use them sparingly and in the correct context. Calling the setStyle() method can be an expensive operation because the call requires notifying all the children of the newly-styled object. The resulting tree of children that must be notified can be quite large.

A common mistake that impacts performance is overusing or unnecessarily using the `setStyle()` method. In general, you only use the `setStyle()` method when you change styles on existing objects. Do not use it when you set up styles for an object for the first time. Instead, set styles in an `<fx:Style>` block, as style properties on the MXML tag, through an external CSS style sheet, or as global styles.

Some applications must call the `setStyle()` method during the application or object instantiation. If this is the case, call the `setStyle()` method early in the instantiation phase. Early in the instantiation phase means setting styles from the component or application's `preinitialize` event, instead of the `initialize` or `creationComplete` event. By setting the styles as early as possible during initialization, you avoid unnecessary style notification and lookup.

If you programmatically create a component and want to set styles on that component, call the `setStyle()` method before you attach it to the display list with a call to the `addElement()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- optimize/CreateStyledButton.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)">
    <fx:Script><![CDATA[
        import spark.components.Button;

        public function initApp(e:Event):void {
            var b:Button = new Button();
            b.label="Click Me";
            b.setStyle("color", 0x00CCFF);
            panel1.addElement(b);
        }
    ]]></fx:Script>

    <s:Panel id="panel1"/>

</s:Application>
```

Setting global styles

Changing global styles (changing a CSS ruleset that is associated with a class or type selector) at run time is an expensive operation. Any time you change a global style, Flash Player must perform the following actions:

- Traverse the entire application looking for instances of that control.
- Check all the control's children if the style is inheriting.

- Redraw that control.

The following example globally changes the Button control’s color style property:

```
<?xml version="1.0"?>
<!-- optimize/ApplyGlobalStyles.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp(event)">
    <fx:Script><![CDATA[
        public function initApp(e:Event):void {
            styleManager.getStyleDeclaration("spark.components.Button").setStyle("color", 
0x00CCFF);
        }
    ]]></fx:Script>

    <s:Panel id="panel1">
        <s:Button id="b1" label="Click Me"/>
        <s:Button id="b2" label="Click Me"/>
        <s:Button id="b3" label="Click Me"/>
    </s:Panel>

</s:Application>
```

If possible, set global styles at authoring time by using CSS. If you must set them at run time, try to set styles by using the techniques described in “Reducing calls to the setStyle() method” on page 2321.

Calling the setStyleDeclaration() and loadStyleDeclarations() methods

The setStyleDeclaration() method is computationally expensive. You can prevent Flash Player from applying or clearing the new styles immediately by setting the `update` parameter to `false`.

The following example sets new class selectors on different targets, but does not trigger the update until the last style declaration is applied:

```
<?xml version="1.0"?>
<!-- optimize/SetStyleDeclarationExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
        private var myButtonStyle:CSSStyleDeclaration =
            new CSSStyleDeclaration('myButtonStyle');
        private var myLabelStyle:CSSStyleDeclaration =
            new CSSStyleDeclaration('myLabelStyle');
        private var myTextAreaStyle:CSSStyleDeclaration =
            new CSSStyleDeclaration('myTextAreaStyle');
        private function initApp():void {
            myButtonStyle.setStyle('color', 'blue');
            myLabelStyle.setStyle('color', 'blue');
            myTextAreaStyle.setStyle('color', 'blue');
        }
        private function applyStyles():void {
           styleManager.setStyleDeclaration("spark.components.Button", myButtonStyle, false);
            styleManager.setStyleDeclaration("spark.components.Label", myLabelStyle, false);
            styleManager.setStyleDeclaration("spark.components.TextArea", myTextAreaStyle,
true);
        }
        ]]>
    </fx:Script>
    <s:Button id="myButton" label="Click Me" click="applyStyles()"/>
    <s:Label id="myLabel" text="This is a label"/>
    <s:TextArea id="myTextArea" text="This is a TextArea"/>
</s:Application>
```

When you pass `false` for the `update` parameter, Flash Player stores the selector but does not apply the style. When you pass `true` for the `update` parameter, Flash Player recomputes the styles for every visual component in the application.

The `loadStyleDeclarations()` method is similarly computationally expensive. When you load a new style sheet, this method triggers an update to the display list by default. You can prevent Flash Player from applying or clearing the new style sheets immediately by setting the `update` parameter to `false`. When you chain calls to `loadStyleDeclarations()` methods, set the update parameter to `false` for all calls except the last one.

Working with containers

Containers provide a hierarchical structure that lets you control the layout characteristics of container children. You can use containers to control child sizing and positioning, or to control navigation among multiple child containers.

When you develop your application, try to minimize the number of containers that you use. This is because most containers provide relative sizing and positioning, which can be resource-intensive operations, especially when an application first starts.

Minimizing container nesting

One common mistake is to create a container that contains a single child. Sometimes having a single child in a container is necessary, such as when you use the container's padding to position the child. But try to identify and remove containers such as these that provide no real functionality. Also keep in mind that the root of an MXML component does not need to be a container.

Another sign of possibly too many containers is when you have a container nested inside another container, where both the parent and child containers have the same type (for example, the HBox or HGroup containers).

It is good practice to avoid deeply nested layouts when possible. For simple applications, if you have nested containers more than three levels deep, you can probably produce the same layout with fewer levels of containers. Deep nesting can lead to performance problems. For larger applications, deeper nesting might be unavoidable.

When you nest containers, each container instance runs measuring and sizing algorithms on its children (some of which are containers themselves, so this measuring procedure can be recursive). When the layout algorithms have processed, and the relative layout values have been calculated, Flash Player draws the complex collection of objects comprising the view. By eliminating unnecessary work at object creation time, you can improve the performance of your application.

Using Grid containers

A Grid container is useful for aligning multiple objects. When you use Grid containers, however, you introduce additional levels of containers with the GridItem and GridRow controls. In many cases, you can achieve the same results by using the HGroup and VGroup containers, and these containers use fewer levels of nesting.

Using absolute layout for containers

You can sometimes improve application start-up time by using containers that support absolute positioning of their children, instead of relative layout containers. Containers that support absolute layout include all Spark containers, plus the MX containers Canvas, Application, and Panel. Containers that only use relative layout include MX containers like Form, HBox, VBox, Grid, and Tile.

All Spark containers support both absolute and relative layout.

Containers that use absolute positioning eliminate the layout logic that relative layout containers use to perform automatic positioning of their children at startup, and replace it with explicit pixel-based positioning. When you use absolute positioning, you must remember to set the *x* and *y* positions of all of the container's children. If you do not set the *x* and *y* positions, the container's children lay out on top of each other at the default *x*, *y* coordinates (0,0).

Container that use absolute positioning are not always more efficient than relative layout containers, however, because they must measure themselves to make sure that they are large enough to contain their children.

Applications that use absolute positioning typically contain a much flatter containment hierarchy. As a result, using these containers can lead to less nesting and fewer overall containers, which improves performance.

Containers that use absolute positioning support constraints, which means that if the container changes size, the children inside the container move with it.

Using absolute sizing

Writing object widths and heights into the code can save time because the Flex layout containers do not have to calculate the size of the object at run time. By specifying container or control widths or heights, you lighten the relative layout container's processing load and subsequently decrease the creation time for the container or control. This technique works with any container or control.

Improving effect performance

Effects let you add animation and motion to your application in response to user or programmatic action. For example, you can use effects to cause a dialog box to bounce slightly when it receives focus, or to slowly fade in when it becomes visible.

Effects can be one of the most processor-intensive tasks performed by an application. Use the techniques described in this section to improve the performance of effects. For more information, see "Introduction to effects" on page 1784.

Increasing effect duration

Increase the duration of your effect with the `duration` property. Doing this spreads the distinct, choppy stages over a longer period of time, which lets the human eye fill in the difference for a smoother effect.

Hiding parts of the target view

Make parts of the target view invisible when the effect starts, play the effect, and then make those parts visible when the effect has completed. To do this, you add logic in the `effectStart` and `effectEnd` event handlers that controls what is visible before and after the effect.

When you apply a Resize effect to a Panel container, for example, the measurement and layout algorithm for the effect executes repeatedly over the duration of the effect. When a Panel container has many children, the animation can be jerky because Flex cannot update the screen quickly enough. Also, resizing one Panel container often causes other Panel containers in the same view to resize.

To solve this problem, you can use the Resize effect's `hideChildrenTargets` property to hide the children of Panel containers while the Resize effect is playing. The value of the `hideChildrenTargets` property is an Array of Panel containers that should include the Panel containers that resize during the animation. When the `hideChildrenTargets` property is `true`, and before the Resize effect plays, Flex iterates through the Array and hides the children of each of the specified Panel containers.

Avoiding bitmap-based backgrounds

Designers often give their views background images that are solid colors with gradients, slight patterns, and so forth. To ease what Flash Player redraws during an effect, try using a solid background color for your background image. Or, if you want a slight gradient instead of a solid color, use a background image that is a SWF or FXG file. These are easier for Flash Player to redraw than standard JPG or PNG files.

Suspending background processing

To improve the performance of effects, you can disable background processing in your application for the duration of the effect by setting the `suspendBackgroundProcessing` property of the Effect to `true`. The background processing that is blocked includes component measurement and layout, and responses to data services for the duration of the effect.

Using the cachePolicy property

An effect can use bitmap caching in Flash Player to speed up animations. An effect typically uses bitmap caching when the target component's drawing does not change while the effect is playing.

The cachePolicy property of UIComponents controls the caching operation of a component during an effect. The `cachePolicy` property can have the following values:

CachePolicy.ON Specifies that the effect target is always cached.

CachePolicy.OFF Specifies that the effect target is never cached.

CachePolicy.AUTO Specifies that Flex determines whether the effect target should be cached. This is the default value.

The cachePolicy property is useful when an object is included in a redraw region but the object does not change. The cachePolicy property provides a wrapper for the cacheAsBitmap property.

Improving rendering speed

The actual rendering of objects on the screen can take a significant amount of time. Improving the rendering times can dramatically improve your application's performance. Use the techniques in this section to help improve rendering speed. In addition, use the techniques described in the previous section, "Improving effect performance" on page 2325, to improve effect rendering speed.

Setting movie quality

You can use the quality property of the wrapper's <object> and <embed> tags to change the rendering of your application in Flash Player. Valid values for the quality property are low, medium, high, autolow, autohigh, and best. The default value is best.

The low setting favors playback speed over appearance and never uses anti-aliasing. The autolow setting emphasizes speed at first but improves appearance whenever possible. The autohigh setting emphasizes playback speed and appearance equally at first, but sacrifices appearance for playback speed if necessary. The medium setting applies some anti-aliasing and does not smooth bitmaps. The high setting favors appearance over playback speed and always applies anti-aliasing. The best setting provides the best display quality and does not consider playback speed. All output is anti-aliased and all bitmaps are smoothed.

For information on these settings, see "About the object and embed tags" on page 2564.

Understanding redraw regions

A *redraw region* is the region around an object that must be redrawn when that object changes. Everything in a redraw region is redrawn during the next rendering phase after an object changes. The area that Flash Player redraws includes the object, and any objects that overlap with the redraw region, such as the background or the object's container.

You can see redraw regions at run time in the debugger version of Flash Player by selecting View > Show Redraw Regions in the player's menu. When you select this option, the debugger version of Flash Player draws red rectangles around each redraw region while the application runs.

By looking at the redraw regions, you can get a sense of what is changing and how much rendering is occurring while your application runs. Flash Player sometimes combines the redraw regions of several objects into a single region that it redraws. As a result, if your objects are spaced close enough together, they might be redrawn as part of one region, which is better than if they are redrawn separately. If the number of regions is too large, Flash Player might redraw the entire screen.

Using the cacheAsBitmap property

To improve rendering speeds, make careful use of the cacheAsBitmap property. You can set this property on any UIComponent.

When you set the cacheAsBitmap property to true, Flash Player stores a copy of the initial bitmap image of an object in memory. If you later need that object, and the object's properties have not changed, Flash Player uses the cached version to redraw the object. This can be faster than using the vectors that make up the object.

Setting the cacheAsBitmap property to true can be especially useful if you use animations or other effects that move objects on the screen. Instead of redrawing the object in each frame during the animation, Flash Player can use the cached bitmap.

The downside is that changing the properties of objects that are cached as bitmaps is more computationally expensive. Each time you change a property that affects the cached object's appearance, Flash Player must remove the old bitmap and store a new bitmap in the cache. As a result, only set the `cacheAsBitmap` property to `true` for objects that do not change much.

Enable bitmap caching only when you need it, such as during the duration of an animation, and only on a few objects at a time because it can be a memory-intensive operation. The best approach might be to change this property at various times during the object's life cycle, rather than setting it once.

Using filters

To improve rendering speeds, do not overuse filters such as DropShadowFilter. The expense of the filter is proportional to the number of pixels in the object that you are applying the filter to. As a result, it is best to use filters on smaller objects. An alternative to the DropShadowFilter is to use the `dropShadowEnabled` property. For more information, see "Using the dropShadowEnabled property" on page 1558.

Using device text

Mixing device text and vector graphics can slow rendering speeds. For example, a DataGrid control that contains both text and graphics inside a cell will be much slower to redraw than a DataGrid that contains just text.

Using clip masks

Using the `scrollRect` and `mask` properties of an object are expensive operations. Try to minimize the number of times you use these properties.

Virtualizing lists and containers

When you use item renderers or list-based controls where not every child is visible on the screen, you should use virtualization to reduce memory usage.

Ideally, you want a parent control to reuse children as the children scroll off the screen, instead of creating a new child for each newly-visible list item. This is the process of virtualization. For example, suppose there is a list of 10 items, but only 5 are visible on the screen at one time. When the user scrolls down to view the sixth item, you want item 1 to be reassigned to item 2, item 2 to be reassigned to item 1, and so on. The result is that only 5 items are in memory at any given time, rather than all 10.

To enable virtualization, use the `useVirtualLayout` property. The default value of this property for list-based controls such as ButtonBar, TabBar, DropDownList, and ComboBox is `true`.

For containers, which wrap item renderers, you can set the virtualization on the layout. The default value of `useVirtualLayout` on layouts such as BasicLayout, VerticalLayout, TileLayout, and HorizontalLayout is `false`. Set this property to `true` to enable virtualization for the containers.

Using large data sets

You can minimize overhead when working with large data sets.

Paging

When you use a DataService class to get your remote data, you might have a collection that does not initially load all of its data on the client. You can prevent large amounts of data from traveling over the network and slowing down your application while that data is processed using paging. The data that you get incrementally is referred to as *paged* data, and the data that has not yet been received is *pending* data.

Paging data using the DataService class provides the following benefits:

- Maximum message size on the destination can be configured.

- If size exceeds the maximum value, multiple message batches are used.
- Client reassembles separate messages.
- Asynchronous data paging across the network.
- User interface elements can display portions of the collection without waiting for the entire collection to load.

For more information, see “Data providers and collections” on page 898.

Disabling live scrolling

Using a DataGrid control with large data sets might make it slow to scroll when using the scrollbar. When the DataGrid displays newly visible data, it calls the `getItemAt()` method on the data provider.

The default behavior of a DataGrid is to continuously update data when the user is scrolling through it. As a result, performance can degrade if you just simply scroll through the data on a DataGrid because the DataGrid is continuously calling the `getItemAt()` method. This can be a computationally expensive method to call.

You can disable this *live scrolling* so that the view is only updated when the scrolling stops by setting the `liveScrolling` property to false.

The default value of the `liveScrolling` property is true. All subclasses of ScrollControlBase, including the MX TextArea, HorizontalList, TileList, and DataGrid classes, have this property.

Dynamically repeating components

There are relative benefits of using List-based controls (rather than the Repeater control) to dynamically repeat components. If you must use the Repeater, however, there are techniques for improving the performance of that control.

Comparing MX List-based controls to the Repeater control

To dynamically repeat components, you can choose between the Repeater or List-based controls, such as HorizontalList, TileList, or List. To achieve better performance, you can often replace layouts you created with a Repeater with the combination of a HorizontalList or TileList and an item renderer.

The Repeater object is useful for repeating a small set of simple user interface components, such as RadioButton controls and other controls typically used in Form containers. You can use the HorizontalList, TileList, or List control when you display more than a few repeated objects.

The HorizontalList control displays data horizontally, similar to the HBox container. The HorizontalList control always displays items from left to right. The TileList control displays data in a tile layout, similar to the Tile container. The TileList control provides a direction property that determines if the next item is down or to the right. The List control displays data in a single vertical column.

Unlike the Repeater object, which instantiates all objects that are repeated, the HorizontalList, TileList, and List controls only instantiate what is visible in the list. The Repeater control takes a data provider (typically an Array) that creates a new copy of its children for each entry in the Array. If you put the Repeater control’s children inside a container that does not use deferred instantiation, your Repeater control might create many objects that are not initially visible.

For example, a VBox container creates all objects within itself when it is first created. In the following example, the Repeater control creates all the objects whether or not they are initially visible:

```
<?xml version="1.0"?>
<!-- optimize/VBoxRepeater.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var imgList:ArrayCollection = new ArrayCollection([
            {img:"butterfly.gif"},
            {img:"butterfly-gray.gif"},
            {img:"butterfly-silly.gif"}
        ]);
    ]]></fx:Script>
    <mx:Panel title="VBox with Repeater">
        <mx:VBox height="150" width="250">
            <mx:Repeater id="r" dataProvider="{imgList}">
                <mx:Image source="../assets/{r.currentItem.img}"/>
            </mx:Repeater>
        </mx:VBox>
    </mx:Panel>
</s:Application>
```

If you use a List-based control, however, Flex only creates those controls in the list that are initially visible. The following example uses the List control to create only the image needed for rendering:

```
<?xml version="1.0"?>
<!-- optimize/ListItems.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
   <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        private static var birdList:Array = ["../assets/butterfly.gif","../assets/butterfly-
gray.gif","../assets/butterfly-silly.gif"];
        [Bindable]
        private var birdListAC:ArrayCollection = new ArrayCollection(birdList);
        private function initCatalog():void {
            birdlist.dataProvider = birdListAC;
        }
    ]]></fx:Script>
    <s:Panel title="List">
        <mx:List id="birdlist"
            rowHeight="150"
            width="250"
            rowCount="1"
            itemRenderer="mx.controls.Image"
            creationComplete="initCatalog()">
        </mx:List>
    </s:Panel>
</s:Application>
```

Using the Repeater control

When using a Repeater control, keep the following techniques in mind:

- Avoid repeating objects that have clip masks because using clip masks is a resource- intensive process.
- Ensure that the containers used as the children of the Repeater control do not have unnecessary container nesting and are as small as possible. If a single instance of the repeated view takes a noticeable amount of time to instantiate, repeating makes it worse. For example, multiple Grid containers in a Repeater object do not perform well because Grid containers themselves are resource-intensive containers to instantiate.
- Set the `recycleChildren` property to `true`. The `recycleChildren` property is a Boolean value that, when set to `true`, binds new data items into existing Repeater children, incrementally creates children if there are more data items, and destroys extra children that are no longer required.

The default value of the `recycleChildren` property is `false` to ensure that you do not leave stale state information in a repeated instance. For example, suppose you use a Repeater object to display photo images and each Image control has an associated NumericStepper control for how many prints you want to order.

Some of the state information, such as the image, comes from the `dataProvider` property. Other state information, such as the print count, is set by user interaction. If you set the `recycleChildren` property to `true` and page through the photos by incrementing the Repeater object's `startingIndex` value, the Image controls bind to the new images, but the NumericStepper control maintains the old information. Use `recycleChildren="false"` only if it is too cumbersome to reset the state information manually, or if you are confident that modifying your `dataProvider` property should not trigger a recreation of the Repeater object's children.

Keep in mind that the `recycleChildren` property has no effect on a Repeater object's speed when the Repeater object loads the first time. The `recycleChildren` property improves performance only for subsequent changes to the Repeater control's data provider. If you know that your Repeater object creates children only once, you do not have to use the `recycleChildren` property or worry about the stale state situation.

Using FXG

If you use FXG to draw graphic elements in your application, or to draw skin parts, you should consider some of the performance issues around how you use it.

When using FXG, do not convert it to MXML graphics and mix the tags inline with your application or other components. Instead, store the FXG file as a separate document. This causes the mxmlc compiler to convert the FXG to Flash primitives rather than map the FXG elements to Flex tags. Flash primitives are much more optimized.

For more information, see "Using FXG in applications built with Flex" on page 1722.

Comparing ArrayLists and ArrayCollections

The ArrayList and ArrayCollection classes store data and provide access to the backing arrays through methods and properties.

The ArrayList class implements the IList interface. The ArrayCollection class implements the IList and ICollectionView interfaces. As a result, the ArrayList class is more lightweight. However, ArrayLists have fewer features than ArrayCollections. ArrayLists do not support:

- Cursors
- Filters
- Sorts

In many cases, you can use an ArrayList instead of an ArrayCollection for data-driven UI components as long as you do not require cursors, filters, and sorts. The following example shows an ArrayList and an ArrayCollection. The control that uses an ArrayList should be faster and use less memory.

```
<?xml version="1.0"?>
<!-- optimize/ArrayListVersusArrayCollection.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:Label text="This ComboBox uses an ArrayList as a data provider:"/>
    <!-- Using an ArrayList is more efficient. -->
   <s:ComboBox>
      <s:ArrayList>
         <fx:String>AK</fx:String>
         <fx:String>AL</fx:String>
         <fx:String>AR</fx:String>
      </s:ArrayList>
   </s:ComboBox>

    <s:Label text="This ComboBox uses an ArrayCollection as a data provider:"/>
   <s:ComboBox>
      <s:ArrayCollection>
         <fx:String>AK</fx:String>
         <fx:String>AL</fx:String>
         <fx:String>AR</fx:String>
      </s:ArrayCollection>
   </s:ComboBox>
</s:Application>
```

In most cases, you cannot get an ArrayList from a web service or remote object call. The results should be set as an ArrayCollection.

Improving charting component performance

You can use various techniques to improve the performance of charting controls.

Avoiding filtering series data

When possible, set the `filterData` property to `false`. In the transformation from data to screen coordinates, the various series types filter the incoming data to remove any missing values that are outside the range of the chart; missing values would render incorrectly if drawn to the screen. For example, a chart that represents vacation time for each week in 2003 might not have a value for the July fourth weekend because the company was closed. If you know your data model will not have any missing values at run time, or values that fall outside the chart's data range, you can instruct a series to explicitly skip the filtering step by setting its `filterData` property to `false`.

Coding the LinearAxis object

If possible, do not let a LinearAxis object autocalculate its range. A LinearAxis control calculating its numeric range can be a resource-intensive calculation. If you know reasonable minimum and maximum values for the range of your LinearAxis, specify them to help your charts render more quickly.

In addition to specifying the range for a LinearAxis, specify an interval (the numeric distance between label values along the axis) value. Otherwise, the chart control must calculate this value.

Coding the CategoryAxis object

Modifying a CategoryAxis object's data provider is more resource intensive than modifying a Series object's data provider. If the data bound to your chart is going to change, but the categories in your chart will stay static, have the CategoryAxis' data provider and Series' data provider refer to different objects. This prevents the CategoryAxis from reevaluating its data provider, which is a resource-intensive computation.

Styling AxisRenderer objects

Improve the rendering time of your AxisRenderers objects by setting particular styles. The AxisRenderers perform many calculations to ensure that they render correctly in all situations. The more help you can give them in restricting their options, the faster they render. Setting the `labelRotation` and `canStagger` styles on the AxisRenderer improve performance. You can set these styles within the tag or in CSS.

Specifying gutter styles

Specify gutter styles when possible. The gutter area of a Cartesian chart is the area between the margins and the actual axis lines. With default values, the chart adjusts the gutter values to accommodate axis decorations. Calculating these gutter values can be resource intensive. By explicitly setting the values of the `gutterLeft`, `gutterRight`, `gutterTop`, and `gutterBottom` style properties, your charts draw quicker and more efficiently.

Using drop shadows

To improve performance, do not use drop-shadows on your series items unless they are necessary. You can selectively add shadows to individual chart series by using renderers such as the ShadowBoxItemRenderer and ShadowLineRenderer classes.

Shadows are implemented as filters in charting controls. As a result, you must remove these shadows by setting the chart control's `seriesFilters` property to an empty Array. The following example removes the shadows from all series, but then changes the renderer for the third series to be a shadow renderer:

```
<?xml version="1.0"?>
<!-- optimize/RemoveShadowsColumnChart.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="600">
    <fx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            {Month: "Jan", Income: 2000, Expenses: 1500, Profit: 500},
            {Month: "Feb", Income: 1000, Expenses: 200, Profit: 800},
            {Month: "Mar", Income: 1500, Expenses: 500, Profit: 1000}
        ]);
    ]]></fx:Script>

    <s:Panel title="Column Chart">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ColumnChart id="myChart" dataProvider="{expenses}">
            <mx:seriesFilters>
                <fx:Array/>
            </mx:seriesFilters>
            <mx:horizontalAxis>
                <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries xField="Month" yField="Income" displayName="Income"/>
                <mx:ColumnSeries xField="Month" yField="Expenses" displayName="Expenses"/>
                <mx:ColumnSeries xField="Month" yField="Profit" displayName="Profit"
                    itemRenderer="mx.charts.renderers.ShadowBoxItemRenderer"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </s:Panel>
</s:Application>
```

Improving startup performance

About startup performance

You could increase the startup time and decrease performance of your applications if you create too many objects or put too many objects into a single view. To improve startup time, minimize the number of objects that are created when the application is first loaded. If a user-interface component is not initially visible at startup, avoid creating that component until you need it. This is called deferred creation. Containers that have multiple views, such as an Accordion container, provide built-in support for this behavior. You can use ActionScript to customize the creation order of multiple-view containers or defer the creation of other containers and controls.

After you improve the *actual* startup time of your application as much as possible, you can improve *perceived* startup time by ordering the creation of containers in the initial view. The default behavior of Flex is to create all containers and their children in the initial view, and then display everything at one time. The user will not be able to interact with the application or see meaningful data until all the containers and their children are created. In some cases, you can improve the user's initial experience by displaying the components in one container before creating the components in the next container. This process is called *ordered creation*.

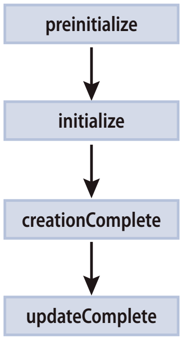
The remaining sections of this topic describe how to use deferred creation to reduce overall application startup time and ordered creation to make the initial startup time appear as short as possible to the user. But before you can fully understand ordered creation and deferred creation, you must also understand the differences between single-view and multiple-view containers, the order of events in a component's startup life cycle, and how to manually instantiate controls from their child descriptors.

About startup order

All Flex components trigger a number of events during their startup procedure. These events indicate when the component is first created, plotted internally, and drawn on the screen. The events also indicate when the component is finished being created and, in the case of containers, when its children are created.

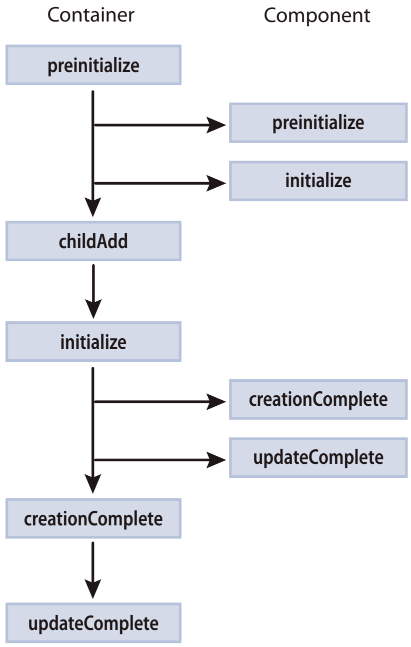
Components are instantiated, added or linked to a parent, and then sized and laid out inside their container. The component creation order is as follows:

The following example shows the major events that are dispatched during a component's creation life cycle:



The creation order is different for containers and components because containers can be the parent of other components or containers. Components within the containers must also go through the creation order. If a container is the parent of another container, the inner container's children must also go through the creation order.

The following example shows the major events that are dispatched during a container's creation life cycle:



After all components are created and drawn, the Application object dispatches an `applicationComplete` event. This is the last event dispatched during an application startup.

The creation order of multiview containers (navigators) is different from standard containers. By default, all top-level views of the navigator are instantiated. However, Flex creates only the children of the initially visible view. When the user navigates to the other views of the navigator, Flex creates those views' children. For more information on the deferred creation of children of multiview containers, see "Using deferred creation" on page 2335.

For a detailed description of the component creation life cycle, see "Using container events" on page 333.

Using deferred creation

By default, containers create only the controls that initially appear to the user. Flex creates the container's other descendants if the user navigates to them. Containers with a single view, such as Group, Box, Form, and Grid containers, create all of their descendants during the container's instantiation because these containers display all of their descendants immediately.

Containers with multiple views, called navigator containers, only create and display the descendants that are visible at any given time. These containers are the MX ViewStack, Accordion, and TabNavigator containers.

When navigator containers are created, they do not immediately create all of their descendants, but only those descendants that are initially visible. Flex defers the creation of descendants that are not initially visible until the user navigates to a view that contains them.

The result of this deferred creation is that an MXML application with navigator containers loads more quickly, but the user experiences brief pauses when he or she moves from one view to another when interacting with the application.

You can instruct each container to create their children or defer the creation of their children at application startup by using the container's `creationPolicy` property. This can improve the user experience after the application loads. For more information, see "About the creationPolicy property" on page 2336.

You can also create individual components whose instantiation is deferred by using the `createDeferredContent()` method (for Spark containers) or the `createComponentsFromDescriptors()` method (for MX containers). For more information, see "Creating deferred components" on page 2339.

About the creationPolicy property

To defer the creation of any component, container, or child of a container, you use the `creationPolicy` property. Every container has a `creationPolicy` property that determines how the container decides whether to create its descendants when the container is created. You can change the policy of a container using MXML or ActionScript.

The valid values for the `creationPolicy` property are `auto`, `all`, and `none`. The meaning of these settings depends on whether the container is a navigator container (multiple-view container) or a single-view container.

The `creationPolicy` property is inheritable when set on the MX Container or Spark SkinnableContainer class. This means that if you set the value of the `creationPolicy` property to `none` on an outer container, all containers within that container inherit that value of the `creationPolicy` property, unless otherwise overridden.

This inheritance does not apply to sibling containers. If you have two containers at the same level (of the same type) and you set the `creationPolicy` of one of them, the other container continues to have the default value of the `creationPolicy` property unless you explicitly set it.

Single-view containers

Single-view containers by default create all their children when the application first starts. You can use the `creationPolicy` property to change this behavior. The following table describes the values of the `creationPolicy` property when you use it with single-view containers:

Value	Description
`all`, `auto`	Creates all controls inside the single-view container. The default value is `auto`, but `all` results in the same behavior.
`none`	Instructs Flex to not instantiate any component within the container until you manually instantiate the controls. When the value of the `creationPolicy` property is `none`, you generally set a width and height for that container explicitly. Normally, Flex scales the container to fit the children that are inside it, but because no children are created, proper scaling is not possible. If you do not explicitly size the container, it grows to accommodate the children when they are created. To manually instantiate controls, you use the `createDeferredContent()` method (for Spark containers) or the `createComponentsFromDescriptors()` method (for MX containers). For more information, see "Creating deferred components" on page 2339.

The following example sets the value of a Spark Panel container's `creationPolicy` property to `auto`, the default value:

```
<?xml version="1.0"?>
<!-- layoutperformance/AutoCreationPolicy.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <s:Panel id="myPanel" creationPolicy="auto">
        <s:Button id="b1" label="Get Weather"/>
    </s:Panel>
</s:Application>
```

The default behavior of all single-view containers is that they and their children are entirely instantiated when the application starts. If you set the `creationPolicy` property to `none`, however, you can selectively instantiate controls within the containers by using the techniques described in “Creating deferred components” on page 2339.

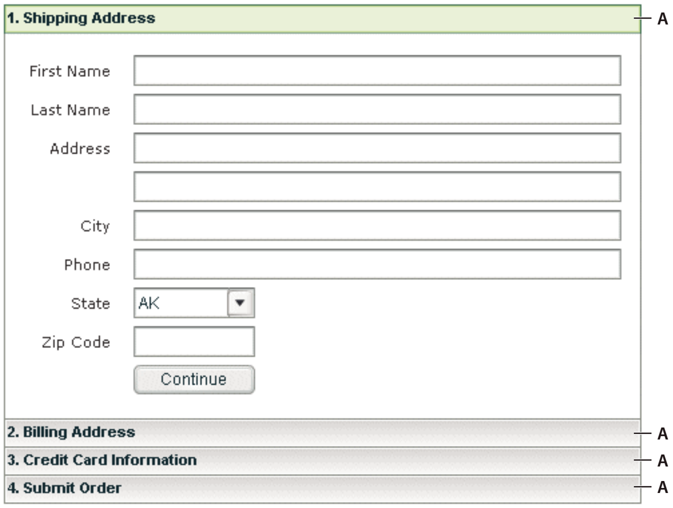
Multiple-view containers

Containers with multiple views, such as the ViewStack and Accordion, do not immediately create all of their descendants, but only those descendants that are visible in the initial view. Flex defers the instantiation of descendants that are not initially visible until the user navigates to a view that contains them. The following containers have multiple views and, so, are defined as navigator containers:

- ViewStack
- TabNavigator
- Accordion

When you instantiate a navigator container, Flex creates all of the top-level children. For example, creating an Accordion container triggers the creation of each of its views, but not the controls within those views. The `creationPolicy` property determines the creation of the child controls inside each view.

When you set the `creationPolicy` property to `auto` (the default value), navigator containers instantiate only the controls and their children that appear in the initial view. The first view of the Accordion container is the initial pane, as the following example shows:



When the user navigates to another panel in the Accordion container, the navigator container creates the next set of controls, and recursively creates the new view’s controls and their descendants. You can use the Accordion container’s `creationPolicy` property to modify this behavior. The following table describes the values of the `creationPolicy` property when you use it with navigator containers:

Value	Description
`all`	Creates all controls in all views of the navigator container. This setting causes a delay in application startup time and an increase in memory usage, but results in quicker response time for user navigation. You should generally not use `all` because of the performance hit that your application will take when it starts up.
`auto`	Creates all controls only in the initial view of the navigator container. This setting causes a faster startup time for the application, but results in slower response time for user navigation. This setting is the default for multiple-view containers.
`none`	Instructs Flex to not instantiate any component within the navigator container or any of the navigator container's panels until you manually instantiate the controls. To manually instantiate controls, you use the `createDeferredContent()` method (for Spark containers) or the `createComponentsFromDescriptors()` method (for MX containers). For more information, see "Creating deferred components" on page 2339.

The following example sets the `creationPolicy` property of an Accordion container to `all`, which instructs the container to instantiate all controls for every panel in the navigator container when the application starts:

```
<?xml version="1.0"?>
<!-- layoutperformance/AllCreationPolicy.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Panel title="Accordion">
        <mx:Accordion id="myAccordion" creationPolicy="all">
            <mx:VBox label="Accordion Button for Panel 1">
                <mx:Label text="Accordion container panel 1"/>
                <s:Button label="Click Me"/>
            </mx:VBox>
            <mx:VBox label="Accordion Button for Panel 2">
                <mx:Label text="Accordion container panel 2"/>
                <s:Button label="Click Me"/>
            </mx:VBox>
            <mx:VBox label="Accordion Button for Panel 3">
                <mx:Label text="Accordion container panel 3"/>
                <s:Button label="Click Me"/>
            </mx:VBox>
        </mx:Accordion>
    </s:Panel>
</s:Application>
```

Setting a container's `creationPolicy` property does not override the policies of the containers within that container.

In the following example, the button1 control is never created because its immediate parent container specifies a `creationPolicy` of `none`, even though the outer container sets the `creationPolicy` property to `all`:

```
<?xml version="1.0"?>
<!-- layoutperformance/TwoCreationPolicies.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Panel title="Outer Panel where creationPolicy=all" creationPolicy="all">
        <s:Panel title="Inner Panel where creationPolicy=none" creationPolicy="none">
            <s:Button id="button1" label="Click Me"/>
        </s:Panel>
    </s:Panel>
</s:Application>
```

Creating deferred components

When you set a container's `creationPolicy` property to `none`, components declared as MXML tags inside that container are not created.

For Spark containers, you use the `createDeferredContent()` method to manually instantiate deferred components. This method is defined on the spark.components.SkinnableContainer class.

For MX containers, you use the `createComponentsFromDescriptors()` method to manually instantiate deferred components. This method is defined on the MX Container base class.

Using the createDeferredContent() method

You use the `createDeferredContent()` method to create deferred child components in a Spark container.

In the following example with a Spark container, the children of the Panel container are not instantiated when the application is first loaded, but only after the user clicks the button:

```
<?xml version="1.0"?>
<!-- layoutperformance/CreationPolicyNoneSpark.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script><![CDATA[
        private function createButtons(e:Event):void {
            myPanel.createDeferredContent();
        }
    ]]></fx:Script>
    <s:Panel id="myPanel" title="Panel with Deferred Content" creationPolicy="none">
        <s:VGroup>
            <s:Button id="b1" label="Hurley"/>
            <s:Button id="b2" label="Jack"/>
            <s:Button id="b3" label="Sawyer"/>
        </s:VGroup>
    </s:Panel>

    <s:Button id="myButton" click="createButtons(event)" label="Create Buttons"/>
</s:Application>
```

Only SkinnableContainer and containers that extend the SkinnableContainer class support deferred instantiation; Groups do not. When a Group is created, its children are always created. You cannot specify a creation policy on a Group.

Using the createComponentsFromDescriptors() method

***Note:** The `createComponentsFromDescriptors()` method is for MX containers. For Spark containers, use the `createDeferredContent()` method.*

You use the `createComponentsFromDescriptors()` method of an MX container to create all the children of a container at one time.

The `createComponentsFromDescriptors()` method has the following signature:

```
container.createComponentsFromDescriptors(recurse:Boolean):Boolean
```

The *recurse* argument determines whether Flex should recursively instantiate children of containers that are children of the top-level container. Set the parameter to `true` to instantiate children of the containers, or `false` to not instantiate the children. The default value is `true`.

On a single-view MX container, calling the `createComponentsFromDescriptors()` method instantiates all controls in that container, regardless of the value of the `creationPolicy` property.

In navigator containers, if you set the `creationPolicy` property to `all`, you do not have to call the `createComponentsFromDescriptors()` method, because the container creates all controls in all views of the container. If you set the `creationPolicy` property to `none` or `auto`, calling the `createComponentsFromDescriptors()` method creates only the current view’s controls and their descendents.

Another common usage is to set the navigator container’s `creationPolicy` property to `auto`. You can then call *navigator*`.getChildAt(n).createComponentsFromDescriptors()` to explicitly create the children of the *n*-th view.

Using the callLater() method

The `callLater()` method queues an operation to be performed for the next screen refresh, rather than in the current update. Without the `callLater()` method, you might try to access a property of a component that is not yet available. The `callLater()` method is most commonly used with the `creationComplete` event to ensure that a component has finished being created before Flex proceeds with a specified method call on that component.

All objects that inherit from the UIComponent class can open the `callLater()` method. It has the following signature:

```
callLater(method:Function, args:Array):void
```

The *method* argument is the function to call on the object. The *args* argument is an optional Array of arguments that you can pass to that function.

The following example defers the invocation of the `doSomething()` method, but when it is opened, Flex passes the Event object and the “Product Description” String in an Array to that function:

```
callLater(doSomething, [event, "Product Description"]);
...
private function doSomething(event:Event, title:String):void {
    ...
}
```

The following example uses a call to the `callLater()` method to ensure that new data is added to a DataGrid before Flex tries to put focus on the new row. Without the `callLater()` method, Flex might try to focus on a cell that does not exist and throw an error:

```
<?xml version="1.0"?>
<!-- layoutperformance/CallLaterAddItem.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    initialize="initData()">
    <fx:Script><![CDATA[
        import mx.collections.*;
        private var DGArray:Array = [
            {Artist:'Pavement', Album:'Charred Walls', Price:11.99},
            {Artist:'Pavement', Album:'Good For You', Price:11.99}];

        [Bindable]
        public var initDG:ArrayCollection;
        //Initialize initDG ArrayCollection variable from the Array.

        public function initData():void {
            initDG=new ArrayCollection(DGArray);
        }

        public function addNewItem():void {
            var o:Object;
            o = {Artist:'Pavement', Album:'Tradio', Price:11.99};
            initDG.addItem(o);
            callLater(focusNewRow);
        }
        public function focusNewRow():void {
            myGrid.editedItemPosition = {
                columnIndex:0,rowIndex:myGrid.dataProvider.length-1
            };
        }

    ]]></fx:Script>
    <s:VGroup>
        <mx:DataGrid id="myGrid" width="350" height="200" dataProvider="{initDG}"
editable="true">
            <mx:columns>
                <fx:Array>
                    <mx:DataGridColumn dataField="Album" />
                    <mx:DataGridColumn dataField="Price" />
                </fx:Array>
            </mx:columns>
        </mx:DataGrid>
        <s:Button id="b1" label="Add New Item" click="addNewItem()"/>
    </s:VGroup>

</s:Application>
```

Another use of the `callLater()` method is to create a recursive method. Because the function is called only after the next screen refresh (or frame), the `callLater()` method can be used to create animations or scroll text with methods that reference themselves.

The following example scrolls ticker symbols across the screen and lets users adjust the speed with an HSlider control:

```
<?xml version="1.0"?>
<!-- layoutperformance/CallLaterTicker.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script><![CDATA[

        [Bindable]
        public var text:String = "SLAR:95.5....TIBA:42....RTF:34.15....AST:23.42";
        [Bindable]
        public var speed:Number = 5;
        public function initTicker():void {
            theText.move( this.width+10, 0 ); // Start the text on the right side.
            callLater(moveText);
        }
        public function moveText():void {
            var xpos:Number = theText.x;
            if( xpos-speed+theText.width < 0 ) {
                xpos = this.width+10; // Start the text on the right side.
            }
            xpos -= speed;
            theText.move(xpos,0);
            callLater(moveText);
        }
        public function changeSpeed():void {
            speed = speedSelector.value;
        }

    ]]></fx:Script>
    <s:Panel title="Ticker Sample" width="400" height="200">
        <s:VGroup>
                <mx:Canvas creationComplete="initTicker()"
                    horizontalScrollPolicy="off" backgroundColor="red" color="white"
                    width="400">
                <mx:Label id="theText" text="{text}" y="0"/>
            </mx:Canvas>
            <mx:HBox>
                <mx:Label text="Speed:"/>
                <mx:HSlider minimum="1" maximum="10" value="{speed}"
                    id="speedSelector" snapInterval="1" tickInterval="1"
                    change="changeSpeed()"/>
            </mx:HBox>
        </s:VGroup>
    </s:Panel>
</s:Application>
```

Ant tasks

The Adobe® Flex® Ant tasks provide a convenient way to build your Flex projects using an industry-standard build management tool. If you are already using Ant projects to build Flex applications, you can use the Flex Ant tasks to replace your `exec` or `java` commands that invoke the mxmlc and compc compilers. If you are not yet using Ant to build your Flex applications, you can take advantage of these custom tasks to quickly and easily set up complex build processes for your Flex applications.

The Ant tasks for Flex include two compiler tasks, `mxmlc` and `compc`. You can use these to compile Flex applications, modules, and component libraries. In addition, the Ant tasks include the `html-wrapper` task that lets you generate custom HTML wrappers and the supporting files for those wrappers, and an `asdoc` Ant task to generate ASDoc documentation.

The `mxmlc` and `compc` Ant tasks extend the `java` Ant task. As a result, you can use all the available attributes of the `java` Ant task in those Ant tasks. This includes `fork`, `maxmemory`, and `classpath`.

More Help topics

The Apache Ant Project

Setting up Flex Ant tasks

Installing the Flex Ant tasks is a simple process. You copy the flexTasks.jar file from a Flex directory to Ant's lib directory. For Adobe® Flex® Builder™, the flexTasks.jar file is located at *flash_builder_install*/sdks/4.6.0/ant/lib. For the SDK, the flexTasks.jar file is located at *sdk_install*/ant/lib.

Copy the flexTasks.jar file to Ant's lib directory (*ant_root*/lib). If you do not copy this file to Ant's lib directory, you must specify it by using Ant's `-lib` option on the command line when you make a project.

For Flash Builder, the ant directory also includes the source code for the Flex Ant tasks.

Using Flex Ant tasks

You can use the Flex Ant tasks in your existing projects or create new Ant projects that use them. There are three tasks that you can use in your Ant projects:

- `mxmlc` — Invokes the application compiler. You use this compiler to compile applications, modules, resource modules, and CSS SWF files.
- `compc` — Invokes the component compiler. You use this compiler to compile SWC files and Runtime Shared Libraries (RSLs).
- `html-wrapper` — Generates the HTML wrapper and supporting files for your application. By using this task, you can select the type of wrapper (with and without deep linking support, with and without Express Install, and with and without Player detection), as well as specify application settings such as the height, width, and background color.
- `asdoc` — Generates ASDoc output for the specified classes in your project.

To use the custom Flex Ant tasks in your Ant projects, you must add the flexTasks.jar file to your project's lib directory, and then point to that JAR file in the `taskdef` task. A `taskdef` task adds a new set of task definitions to your current project. You use it to add task definitions that are not part of the default Ant installation. In this case, you use the `taskdef` task to add the `mxmlc`, `compc`, and `html-wrapper` task definitions to your Ant installation. In addition, for most projects you set the value of the `FLEX_HOME` variable so that Ant can find your flex-config.xml file and so that you can add the frameworks directory to your source path.

Use the Flex tasks in Ant:

1 Add a new `taskdef` task to your project. In this task, specify the flexTasks.tasks file as the resource, and point to the flexTasks.jar file for the classpath. For example:

```
<taskdef resource="flexTasks.tasks" classpath="${basedir}/flexTasks/lib/flexTasks.jar"/>
```

2 Define the `FLEX_HOME` and `APP_ROOT` properties. Use these properties to point to your Flex SDK's root directory and application's root directory. Although not required, creating properties for these directories is a common practice because you will probably use them several times in your Ant tasks. For example:

```
<property name="FLEX_HOME" value="C:/flex/sdk"/>
<property name="APP_ROOT" value="myApps"/>
```

3 Write a target that uses the Flex Ant tasks. The following example defines the `main` target that uses the `mxmlc` task to compile the Main.mxml file:

```
<target name="main">
    <mxmlc file="${APP_ROOT}/Main.mxml" keep-generated-actionscript="true">
        <load-config filename="${FLEX_HOME}/frameworks/flex-config.xml"/>
        <source-path path-element="${FLEX_HOME}/frameworks"/>
    </mxmlc>
</target>
```

The following shows the complete example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- mySimpleBuild.xml -->
<project name="My App Builder" basedir=".">
    <taskdef resource="flexTasks.tasks"
        classpath="${basedir}/flexTasks/lib/flexTasks.jar"/>
    <property name="FLEX_HOME" value="C:/flex/sdk"/>
    <property name="APP_ROOT" value="myApp"/>
    <target name="main">
        <mxmlc file="${APP_ROOT}/Main.mxml" keep-generated-actionscript="true">
            <load-config filename="${FLEX_HOME}/frameworks/flex-config.xml"/>
            <source-path path-element="${FLEX_HOME}/frameworks"/>
        </mxmlc>
    </target>
</project>
```

This example shows how to use different types of options for the `mxmlc` task. You can specify the value of the `keep-generated-actionscript` option as an attribute of the `mxmlc` task's tag because it does not take any child tags. To specify the values of the `load-config` and `source-path` options, you create child tags of the `mxmlc` task's tag. For more information on using options, see "Working with compiler options" on page 2345.

4 Execute the Ant project, as shown in the following example:

```
> ant -buildfile mySimpleBuild.xml main
```

If you did not copy the flexTasks.jar file to Ant's lib directory as described in "Setting up Flex Ant tasks" on page 2343, you must include the JAR file by using Ant's `-lib` option. For example:

```
> ant -lib c:/ant/lib/flexTasks.jar -buildfile mySimpleBuild.xml main
```

The output of these commands should be similar to the following:

```
Buildfile: mySimpleBuild.xml
main:
    [mxmlc] Loading configuration file C:\flex\sdk\frameworks\flex-config.xml
    [mxmlc] C:\myfiles\flex4\ant_tests\apps\Main.swf (150035 bytes)
BUILD SUCCESSFUL
Total time: 10 seconds
>
```

To embed fonts in your Flex ant tasks, you must be sure to include the font manager in your build file. For more information, see Embedding Fonts Using AS3 and Flex 4 Ant Tasks.

Working with compiler options

The compc and mxmlc compilers share a very similar set of options. As a result, the behavior of the `mxmlc` and `compc` Ant tasks are similar as well.

You can specify options for the `mxmlc` and `compc` Flex tasks in a number of ways:

- Task attributes
- Single argument options
- Multiple argument options
- Nested elements
- Implicit FileSets

Task attributes

The simplest method of specifying options for the Flex Ant tasks is to specify the name and value of command-line options as a task attribute. In the following example, the `file` and `keep-generated-actionscript` options are specified as attributes of the `mxmlc` task:

```
<mxmlc file="${APP_ROOT}/Main.mxml" keep-generated-actionscript="true">
```

Many mxmlc and compc options have aliases (alternative shorter names). The Flex Ant tasks support all documented aliases for these options.

Single argument options

Many compiler options must specify the value of one or more parameters. For example, the `load-config` option takes a parameter named `filename`.

To specify the option in an Ant task, you add a child tag with that option as the tag name. You set the values of the arguments by including an attribute in the tag whose name is the option name and whose value is the value of the argument.

The following example sets the values of the `load-config` and `source-path` options, which have arguments named `filename` and `path-element` respectively:

```
<mxmlc ... >
    <load-config filename="${FLEX_HOME}/frameworks/flex-config.xml"/>
    <source-path path-element="${FLEX_HOME}/frameworks"/>
</mxmlc>
```

The `source-path` option can take more than one `path-element` parameter. For more information, see “Repeatable options” on page 2346.

You can see which options take parameters by looking at the mxmlc command-line syntax. In the Flex bin directory, enter the following command:

```
mxmlc -help list advanced
```

The output includes all options for the mxmlc compiler. The syntax for the `load-config` option appears like the following:

```
-load-config <filename>
```

This indicates that the `load-config` option takes a single argument called `filename`.

Multiple argument options

Some compiler options, such as the `default-size` option, take more than one argument. The `default-size` option takes a `height` and a `width` argument. You set the values of options that take multiple arguments by using a nested element of the same name, with the attributes of the element corresponding to the arguments of the option as shown in the command-line compiler's online help.

For example, the online help for mxmlc shows the following syntax for the `default-size` option:

```
-default-size <width> <height>
```

To pass the option `-default-size 800 600` to the `mxmlc` task, use the following syntax:

```
<mxmlc ...>
    <default-size width="800" height="600"/>
</mxmlc>
```

Repeatable options

Some compiler options are repeatable. The online help shows their arguments in square brackets, followed by a bracketed ellipses, like this:

```
-compiler.source-path [path-element] [...]
```

You set the value of repeatable options by using multiple nested elements of the same name as the option, along with attributes of the same name as they appear in the online help.

The following example sets two values for the `compiler.source-path` option:

```
<mxmlc ...>
    <compiler.source-path path-element="src"/>
    <compiler.source-path path-element="../bar/src"/>
</mxmlc>
```

Nested elements

In some situations, options that are closely related are grouped together in a nested element of the main task element. For example, the command-line compiler options with the `compiler.fonts` and `metadata` prefixes can be grouped into nested elements. The `compiler.fonts` options use the element name `fonts` and the `metadata` options use the element name `metadata`.

The following example shows how to use the `metadata` nested element:

```
<mxmlc ...>
    <metadata description="foo app">
        <contributor name="Joe" />
        <contributor name="Nick" />
    </metadata>
</mxmlc>
```

In this example, you drop the `metadata` prefix when setting the `description` and `contributor` options as a nested element.

This is a uniformly applied rule with one exception: the `compiler.fonts.languages.language-range` option is set using a nested element with the name `language-range`, rather than `languages.language-range`.

Implicit FileSets

There are many examples in the Apache Ant project where tasks behave as implicit FileSets. For example, the `delete` task, while supporting additional attributes, supports all of the attributes (such as `dir` and `includes`) and nested elements (such as `include` and `exclude`) of a FileSet to specific the files to be deleted.

Some Flex Ant tasks allow nested attributes that are implicit FileSets. These nested attributes support all the attributes and nested elements of an Ant FileSet while adding additional functionality. These elements are usually used to specify repeatable arguments that take a filename as an argument.

The following example uses the implicit FileSet syntax with the `include-sources` nested element:

```
<include-sources dir="player/avmplus/core"
    includes="builtin.as, Date.as, Error.as, Math.as, RegExp.as, XML.as"/>
```

When a nested element in a Flex Ant task is an implicit FileSet, it supports one additional attribute: `append`. The reason for this is that some repeatable options for the Flex compilers have default values. When setting these options in a command line, you can append new values to the default value of the option, or replace the default value with the specified values. Setting the `append` value to `true` adds the new option to the list of existing options. Setting the `append` value to `false` replaces the existing options with the new option. By default, the value of the `append` attribute is `false` in implicit FileSets.

The following example sets the value of the `append` attribute to `true` and uses the Ant `include` element of an implicit FileSet to add multiple SWC files to the `include-libraries` option:

```
<compiler.include-libraries dir="${swf.output}" append="true">
    <include name="MyComponents.swc" />
    <include name="AcmeComponents.swc" />
    <include name="DharmaComponents.swc" />
</compiler.include-libraries>
```

The following options are implemented as implicit FileSets:

```
compiler.external-library-path
compiler.include-libraries
compiler.library-path
compiler.theme
compiler.include-sources (compc only)
```

The Flex Ant task's implicit FileSets are also different from the Ant project's implicit FileSets in that they support being empty, as in the following example:

```
<external-library-path/>
```

This is equivalent to using `external-library-path=` on the command line.

Using the mxmlc task

You use the `mxmlc` Flex Ant task to compile applications, modules, resource modules, and Cascading Style Sheets (CSS) SWF files. This task supports most mxmlc command-line compiler options, including aliases.

For more information on using the mxmlc command-line compiler, see “Using mxmlc, the application compiler” on page 2174.

Required attributes

The mxmlc task requires the file attribute. The file attribute specifies the MXML file to compile. This attribute does not have a command-line equivalent because it is the default option on the command line.

Unsupported options

The following mxmlc command-line compiler options are not supported by the mxmlc task:

- help
- version

Example

The following example mxmlc task explicitly defines the source-path and library-path options, in addition to other properties such as incremental and keep-generated-actionscript. This example also specifies an output location for the resulting SWF file. This example defines two targets: main and clean. The main target compiles the Main.mxml file into a SWF file. The clean target deletes the output of the main target.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- myMXMLCBuild.xml -->
<project name="My App Builder" basedir=".">
    <taskdef resource="flexTasks.tasks" classpath="${basedir}/flexTasks/lib/flexTasks.jar" />
    <property name="FLEX_HOME" value="C:/flex/sdk"/>
    <property name="APP_ROOT" value="apps"/>
    <property name="DEPLOY_DIR" value="c:/jrun4/servers/default/default-war"/>
    <target name="main">
        <mxmlc
            file="${APP_ROOT}/Main.mxml"
            output="${DEPLOY_DIR}/Main.swf"
            actionscript-file-encoding="UTF-8"
            keep-generated-actionscript="true"
            incremental="true">
            <!-- Get default compiler options. -->
            <load-config filename="${FLEX_HOME}/frameworks/flex-config.xml"/>

            <!-- List of path elements that form the roots of ActionScript
            class hierarchies. -->
            <source-path path-element="${FLEX_HOME}/frameworks"/>

            <!-- List of SWC files or directories that contain SWC files. -->
            <compiler.library-path dir="${FLEX_HOME}/frameworks" append="true">
                <include name="libs" />
                <include name="../bundles/{locale}" />
            </compiler.library-path>

            <!-- Set size of output SWF file. -->
            <default-size width="500" height="600" />
        </mxmlc>
    </target>
    <target name="clean">
        <delete dir="${APP_ROOT}/generated"/>
        <delete>
            <fileset dir="${DEPLOY_DIR}" includes="Main.swf"/>
        </delete>
    </target>
</project>
```

Using the compc task

You use the `compc` Flex Ant task to compile component SWC files. This task supports most compc command-line compiler options, including aliases. For more information on using the compc command-line compiler, see "Using compc, the component compiler" on page 2194.

Required attributes

The only required attribute for the `compc` task is the `output` attribute, which specifies the name of the SWC file that the `compc` task creates.

Special attributes

The `include-classes` attribute takes a space-delimited list of class names. For example:

```
<compc include-classes="custom.MyPanel custom.MyButton" ... >
    ...
</compc>
```

When using the `include-resource-bundles` attribute, you should not specify them as a comma or space-delimited list in a single entry. Instead, add a separate child tag for each resource bundle that you want to include, as the following example shows:

```
<compc output="${swf.output}/compc_rb.swc" locale="en_US">
    <include-resource-bundles bundle="ErrorLog"/>
    <include-resource-bundles bundle="LabelResource"/>
    <sp path-element="locale/{locale}"/>
</compc>
```

Unsupported options

The following compc command-line compiler options are not supported by the `compc` task:

- `help`
- `version`

Example

The following example `compc` task builds a new SWC file that contains two custom components and other assets. The components are added to the SWC file by using the `include-classes` attribute. The source files for the components are in a subdirectory called components. The other assets, including four images and a CSS file, are added to the SWC file by using the `include-file` element.

This example defines two targets: `main` and `clean`. The main target compiles the MyComps.swc file. The `clean` target deletes the output of the main target.

```
<?xml version="1.0" encoding="utf-8"?>
<project name="My Component Builder" basedir=".">
    <taskdef resource="flexTasks.tasks" classpath="${basedir}/flexTasks/lib/flexTasks.jar" />
    <property name="FLEX_HOME" value="C:/flex/sdk"/>
    <property name="DEPLOY_DIR" value="c:/jrun4/servers/default/default-war"/>
    <property name="COMPONENT_ROOT" value="components"/>
    <target name="main">
        <compc
            output="${DEPLOY_DIR}/MyComps.swc"
            include-classes="custom.MyButton custom.MyLabel">
            <source-path path-element="${basedir}/components"/>
            <include-file name="f1-1.jpg" path="assets/images/f1-1.jpg"/>
            <include-file name="f1-2.jpg" path="assets/images/f1-2.jpg"/>
            <include-file name="f1-3.jpg" path="assets/images/f1-3.jpg"/>
            <include-file name="f1-4.jpg" path="assets/images/f1-4.jpg"/>
            <include-file name="main.css" path="assets/css/main.css"/>
        </compc>
    </target>
    <target name="clean">
        <delete>
            <fileset dir="${DEPLOY_DIR}" includes="MyComps.swc"/>
        </delete>
    </target>
</project>
```

Using the html-wrapper task

The `html-wrapper` Ant task generates files that you deploy with your applications. An HTML wrapper consists of a generated HTML page and a JavaScript file that defines embed logic. The output can also include files that support features such as deep linking and Express Install.

The `html-wrapper` task outputs the index.html and swfobject.js files for your application. The swfobject.js file defines the SWFObject 2 logic that embeds SWF files in HTML.

If you enable deep linking support, the `html-wrapper` task also outputs the deep linking files such as historyFrame.html, history.css, and history.js. If you enable express installation, the `html-wrapper` task also outputs the playerProductInstall.swf file.

You typically deploy these files, along with your application's SWF file, to a web server. Users request the HTML wrapper, which embeds the SWF file. You can customize the output of the wrapper and its supporting files after it is generated by Ant.

For more information on the HTML wrapper, see “Creating a wrapper” on page 2552.

Supported attributes

The attributes of the `html-wrapper` task correspond to some of the arguments in the `swfobject.embedSWF()` method of the default HTML wrapper.

The following table describes the supported attributes of the `html-wrapper` task:

Attribute	Description
`application`	The name of the SWF object in the HTML wrapper. You use this name to refer to the SWF object in JavaScript or when using the ExternalInterface API. This value should not contain any spaces or special characters. This attribute sets the value of the `<embed>` tag's `name` attribute and the `<object>` tag's `id` attribute.
`bgcolor`	Specifies the background color of the application. Use this property to override the background color setting specified in the SWF file. This property does not affect the background color of the HTML page. This attribute sets the value of the `params.bgcolor` argument in the `embedSWF()` method. The default value is `white`.
`express-install=false|true`	Determines whether to include Express Install logic in the HTML wrapper. When set to `true`, the Ant task copies the playerProductInstall.swf file to the output directory and referenced in the wrapper file. If you set this option to `true`, the `version-detection` option is also assumed to be `true`. For more information, see "Using Express Install in the wrapper" on page 2558. The default value is `false`.
`file`	Sets the file name of the HTML output file. The default value is "index.html".
`height`	Defines the height, in pixels, of the SWF file. Adobe® Flash® Player makes a best guess to determine the height of the application if none is provided. This attribute sets the value of the third argument in the `embedSWF()` method. The default value is 400.
`history`	Set to `true` to include deep linking support (also referred to as *history management*) in the HTML wrapper. Set to `false` to exclude deep linking from the wrapper. When you set this attribute to true, Ant creates a history directory and stores the historyFrame.html, history.css, and history.js files in it. The default value is `false`. For more information on deep linking, see "Deep linking" on page 2022.
`output`	Sets the directory that Ant writes the generated files to. If you do not set the value of this option, Ant creates the wrapper files in the same directory as the build file.
`swf`	Sets the name of the SWF file that the HTML wrapper embeds (for example, Main). Do not include the *.swf extension; the extension is appended to the name for you. This attribute sets the value of the attributes.name and attributes.id arguments in the `embedSWF()` method. It also sets the value of the first argument in that method call. This SWF file does not have to exist when you generate the HTML wrapper. It is used by SWFObject 2 to point to the location of the SWF file at deployment time.
`title`	Sets the value of the `<title>` tag in the head of the HTML page. The default value is `Flex Application`.
`version-detection=true|false`	Determines whether to include version detection logic in the wrapper. Set this value to `false` to disable all Player version logic in the HTML wrapper. The default value is `true`.
`version-major`	Sets the value of the `swfVersionStr` variable in the HTML wrapper. The default value is 10. The value of this attribute only matters if you include version detection in your wrapper by setting the `template` attribute to `express-installation` or `client-side-detection`.

Attribute	Description
`version-minor`	Sets the value of the `swfVersionStr` variable in the HTML wrapper. The default value is 0. The value of this attribute only matters if you include version detection in your wrapper by setting the `template` attribute to `express-installation` or `client-side-detection`.
`version-revision`	Sets the value of the `swfVersionStr` variable in the HTML wrapper. The default value is 0. The value of this attribute only matters if you include version detection in your wrapper by setting the `template` attribute to `express-installation` or `client-side-detection`.
`width`	Defines the width, in pixels, of the SWF file. Flash Player makes a best guess to determine the width of the application if none is provided. This attribute sets the value of the fourth argument in the `embedSWF()` method. The default value is 400.

Required attributes

The `html-wrapper` task requires the `swf` attribute. If you specify only the `swf` attribute, the default wrapper will have the following settings:

```
title="Flex Application"
swfVersionStr = "10.0.0";
params.quality = "high";
params.bgcolor = "white";
params.allowscriptaccess = "sameDomain";
params.allowfullscreen = "true";
attributes.align = "middle";
height="400"
width="400"
```

Be sure to use the filename only and not the filename plus extension when specifying the value of the `swf` attribute. The `html-wrapper` task appends `.swf` to the end of its value before passing it to the `embedSWF()` method.

For example, do this:

```
swf="Main"
```

Do not do this:

```
swf="Main.swf"
```

Unsupported options

The `html-wrapper` task does not support all properties used by SWFObject 2 to embed a SWF file in an HTML wrapper. Parameters you cannot set include `quality`, `allowScriptAccess`, `classid`, `pluginspage`, and `type`.

Example

The following example project includes a wrapper target that uses the `html-wrapper` task to generate a wrapper with deep linking, Player detection, and Express Install logic. This target (`wrapper`) also sets the height and width of the SWF file. The project also includes a `clean` target that deletes all the files generated by the wrapper target.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- myWrapperBuild.xml -->
<project name="My Wrapper Builder" basedir=".">
    <taskdef resource="flexTasks.tasks" classpath="${basedir}/lib/flexTasks.jar"/>
    <property name="FLEX_HOME" value="C:/p4/flex/flex/sdk"/>
    <property name="APP_ROOT" value="c:/temp/ant/wrapper"/>
    <target name="wrapper">
        <html-wrapper
            title="Welcome to My Flex App"
            file="index.html"
            height="300"
            width="400"
            application="app"
            swf="Main"
            history="true"
            express-install="true"
            version-detection="true"
            output="${APP_ROOT}"/>
    </target>
    <target name="clean">
        <delete>
            <!-- Deletes playerProductInstall.swf -->
            <fileset dir="${APP_ROOT}"
                includes="playerProductInstall.swf"
                defaultexcludes="false"/>
            <!-- Deletes the swfobject.js file -->
            <fileset dir="${APP_ROOT}" includes="*.js" defaultexcludes="false"/>
            <!-- Deletes the previously-generated HTML wrapper file -->
            <fileset dir="${APP_ROOT}" includes="*.html" defaultexcludes="false"/>
            <!-- Deletes the history files -->
            <fileset dir="${APP_ROOT}/history" includes="*.*" defaultexcludes="false"/>
        </delete>
    </target>
</project>
```

Using the asdoc task

The Flex Ant tasks include a task to run the asdoc utility. This utility generates HTML documentation based on a defined set of tags that you can include in your ActionScript and MXML files.

For more information about using ASDoc, see “ASDoc” on page 2241.

Required attributes

The asdoc Ant task requires that you specify a target class or classes to generate ASDoc for. This must be either a list of one or more classes, a directory of classes, or a namespace. You specify these by using the `doc-classes`, `doc-sources`, or `doc-namespaces` attributes.

If you use the `doc-classes` or `doc-namespaces` attributes, you must also specify a `source-path` argument. If you use the doc-namespaces attribute, you must also specify a `namespace` attribute.

Adobe recommends that you specify a value for the `output` attribute. If you do not specify a value, Ant creates a directory called asdoc-output in the same directory that you launched the command from. Ant then stores the HTML output files in that new directory.

Unsupported options

All options of the asdoc utility are supported by the `asdoc` Ant task.

Example

The following example defines two targets, `doc` and `clean`. The `doc` target generates ASDoc for the Spark and MX button classes. The `clean` target deletes all the files generated by the `doc` target.

```
<?xml version="1.0" encoding="utf-8"?>
<project name="ASDoc Builder" basedir=".">
    <property name="FLEX_HOME" value="C:/p4/flex/flex/sdk"/>
    <property name="OUTPUT_DIR" value="C:/temp/ant/asdoc"/>
    <taskdef resource="flexTasks.tasks" classpath="${FLEX_HOME}/ant/lib/flexTasks.jar" />
    <target name="doc">
        <asdoc output="${OUTPUT_DIR}" lenient="true" failonerror="true">
            <doc-sources
                path-
element="${FLEX_HOME}/frameworks/projects/spark/src/spark/components/Button.as"/>
            <doc-sources
                path-
element="${FLEX_HOME}/frameworks/projects/framework/src/mx/controls/Button.as"/>
        </asdoc>
    </target>
    <target name="clean">
        <delete includeEmptyDirs="true">
            <fileset dir="${OUTPUT_DIR}" includes="**/*"/>
        </delete>
    </target>
</project>
```

You will have to change the values of the Flex home and output directory to match the locations of the directories on your file system.

To run this example, you can use the following Ant command:

```
ant clean doc
```

This example generates simple ASDoc output for the Spark and MX Button controls by using the `doc-sources` attribute. The output does not include all the parent classes that provide the inherited methods, properties, events, and other members of the Button controls.

To generate ASDoc that includes the parent classes of the Button controls, you can use an Ant task that uses the `doc-classes` attribute. When you do this, you must be sure to also define the `source-path` for the classes.

The following example generates ASDoc for the MX Button control. It includes the parent classes that this control inherits its members from.

```
<?xml version="1.0" encoding="utf-8"?>
<project name="ASDoc Builder" basedir=".">
    <property name="FLEX_HOME" value="C:/p4/flex/flex/sdk"/>
    <property name="OUTPUT_DIR" value="C:/temp/ant/asdoc"/>
    <taskdef resource="flexTasks.tasks" classpath="${FLEX_HOME}/ant/lib/flexTasks.jar" />
    <target name="doc">
        <asdoc output="${OUTPUT_DIR}" lenient="true" failonerror="true">
            <compiler.source-path
                path-element="${FLEX_HOME}/frameworks/projects/framework/src"/>
            <doc-classes class="mx.controls.Button"/>
        </asdoc>
    </target>
    <target name="clean">
        <delete includeEmptyDirs="true">
            <fileset dir="${OUTPUT_DIR}" includes="**/*"/>
        </delete>
    </target>
</project>
```

You can also define an `asdoc` Ant task that includes all components in a particular namespace by using the `doc-namespaces` attribute. When you want to document all classes in a namespace, you must also point to the manifest file for that namespace by using the `namespace` attribute.

The following example generates ASDoc for all components in the Spark namespace:

```
<?xml version="1.0" encoding="utf-8"?>
<project name="ASDoc Builder" basedir=".">
    <property name="FLEX_HOME" value="C:/p4/flex/flex/sdk"/>
    <property name="OUTPUT_DIR" value="C:/temp/ant/asdoc"/>
    <taskdef resource="flexTasks.tasks" classpath="${FLEX_HOME}/ant/lib/flexTasks.jar" />
    <target name="doc">
        <asdoc output="${OUTPUT_DIR}" lenient="true" failonerror="true">
            <compiler.source-path
                path-element="${FLEX_HOME}/frameworks/projects/spark/src"/>
            <doc-namespaces uri="library://ns.adobe.com/flex/spark"/>
            <namespace
                uri="library://ns.adobe.com/flex/spark"
                manifest="${FLEX_HOME}/frameworks/projects/spark/manifest.xml"/>
        </asdoc>
    </target>
    <target name="clean">
        <delete includeEmptyDirs="true">
            <fileset dir="${OUTPUT_DIR}" includes="**/*"/>
        </delete>
    </target>
</project>
```

In this example, only the members that are inherited from other classes in the Spark namespace are added to the ASDoc output.

The flex-config.xml file contains a list of predefined namespace URIs and the locations of their manifest files. As a result, if you use the `asdoc` Ant task to generate ASDoc for one of these namespaces, and not a custom namespace, then you are not required to define the location of the manifest file. This is because the `asdoc` Ant task reads in their definitions. It is, however, best practice to define the manifest file location for all namespaces.

Chapter 10: Custom components

Custom Flex components

Adobe® Flex® supports a component-based development model. You use the predefined components included with Flex to build your applications, and create components for your specific application requirements. You can create custom components by using MXML or ActionScript.

Flex Developer Mike Jones has written the following book about creating custom Flex components: Developing Flex 4 Components: Using ActionScript 3.0 and MXML to Extend Flex and AIR Applications.

About custom components

Defining your own custom components has several benefits. One advantage is that components let you divide your applications into modules that you can develop and maintain separately. By implementing commonly used logic within custom components, you can also build a suite of reusable components that you can share among multiple Flex applications.

Also, you can extend the Flex class hierarchy to base your custom components on the set of predefined Flex components. You can create custom versions of Flex visual controls, as well as custom versions of nonvisual components, such as validators, formatters, and effects.

You can build an entire Flex application in a single MXML file that contains both your MXML code and any supporting ActionScript code. However, as your application gets larger, your single file also grows in size and complexity. This type of application would soon become difficult to understand and debug, and very difficult for multiple developers to work on simultaneously. By breaking it up into components, you can simplify the architecture and divide the application among multiple developers.

Using modules in application development

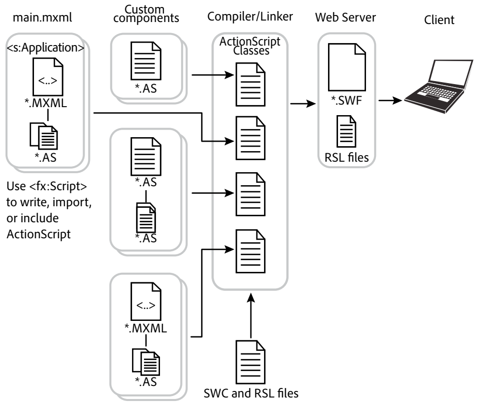
A common coding practice is to divide an application into functional units, or modules, where each module performs a discrete task. Dividing your application into modules provides you with many benefits, including the following:

Ease of development Different developers or development groups can develop and debug modules independently of each other.

Reusability You can reuse modules in different applications so that you do not have to duplicate your work.

Maintainability By developing your application in discrete modules, you can isolate and debug errors faster than you could if you developed your application in a single file.

In Flex, a module corresponds to a custom component, implemented either in MXML or in ActionScript. The following image shows an example of a Flex application divided into components:



This example shows the following relationships among the components:

- You define a main MXML file that contains the `<s:Application>` tag.
- In your main MXML file, you define an ActionScript block that uses the `<fx:Script>` tag. Inside the ActionScript block, you write ActionScript code, or include external logic defined by an ActionScript file. Typically, you use this area to write small amounts of ActionScript code. If you must write large amounts of ActionScript code, you should include an external file.
- The main MXML file uses MXML and ActionScript to reference components supplied with Flex, and to reference your custom components.
- Custom components can reference other custom components.

The Flex class hierarchy

Flex is implemented as an ActionScript class hierarchy. That class hierarchy contains component classes, manager classes, data-service classes, and classes for all other Flex features. For a complete description of the class hierarchy, see the ActionScript 3.0 Reference for the Adobe Flash Platform.

All visual components are derived from the UIComponent ActionScript class. Flex nonvisual components are also implemented as a class hierarchy in ActionScript. The most commonly used nonvisual classes are the Validator, Formatter, and Effect base classes.

You create custom components by extending the Flex class hierarchy using the MXML and ActionScript languages. Components inherit the properties, methods, events, styles, and effects of their superclasses.

Spark and MX components

Flex defines two sets of components: Spark and MX. The Spark components are new for Flex 4 and are defined in the spark.* packages. The MX components shipped in previous releases of Flex and are defined in the mx.* packages.

Spark and MX define some of the same components. For example, Spark defines a button control in the spark.components package, and MX defines a button control in the mx.controls package. When a component is available in both Spark and MX, Adobe recommends that you use the Spark component.

Spark and MX also define components that are unique. For example, Spark defines components to perform three dimensional effects. MX defines charting controls not included in Spark. Your applications often contains a mixture of Spark and MX components.

Customizing existing components

One reason for you to create a component is to customize an existing component for your application requirements. This customization could be as simple as setting the `label` property of a Button control to *Submit* to create a custom button for all of your forms.

You might also want to modify the behavior of a Flex component. For example, a VBox container lays out its children from the top of the container to the bottom in the order in which you define the children within the container. Instead, you might want to customize the VBox container to lay out its children from bottom to top.

Another reason to customize a Flex component is to add logic or behavior to it. For example, you might want to modify the TextInput control so that it supports a key combination to delete all the text entered into the control. Or, you might want to modify a component so that it dispatches a new event type when a user carries out an action.

To create your own components, you create subclasses from the UIComponent class, or any other class in the Flex component hierarchy. For example, if you want to create a component that behaves almost the same as a Button component does, you can extend the Button class instead of recreating all the functionality of the Button class from the base classes.

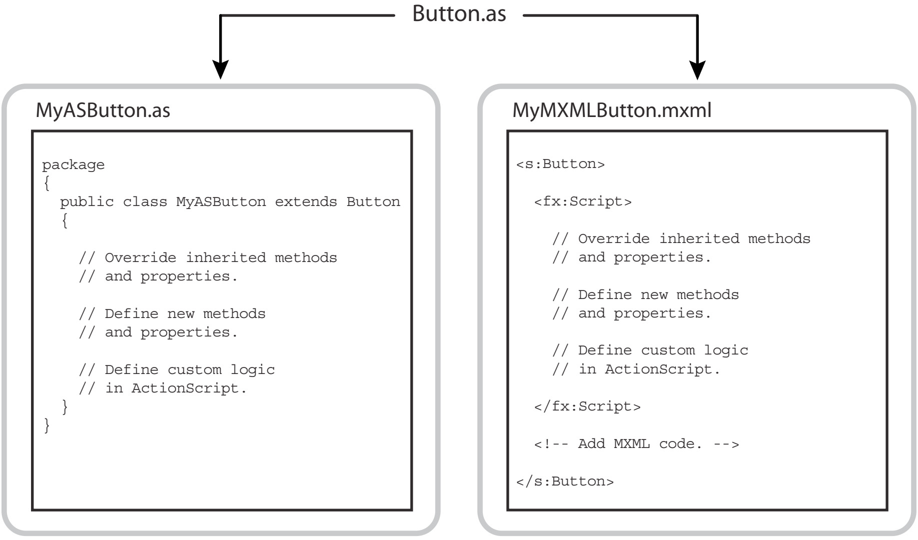
Depending on the modifications that you want to make, you can create a subclass of a Flex component in MXML or ActionScript.

The relationship between MXML components and ActionScript components

To create a custom component in ActionScript, you create a subclass from a class in the Flex class hierarchy. The name of your class (for example, MyASButton), must correspond to the name of the ActionScript file; for example, MyASButton.as. The subclass inherits all of the properties and methods of the superclass. In this example, you use the `<MyASButton>` tag to reference it in MXML.

When you create a custom component in MXML, the Flex compiler automatically creates an ActionScript class. The name of the MXML file (for example, MyMXMLButton.mxml) corresponds to the ActionScript class name. In this example, the ActionScript class is named MyMXMLButton, and you use the `<MyMXMLButton>` tag to reference it in MXML.

The following example shows two components based on the Flex Button component, one defined in ActionScript and the other in MXML:



```
package
{
  public class MyASButton extends Button
  {

    // Override inherited methods
    // and properties.

    // Define new methods
    // and properties.

    // Define custom logic
    // in ActionScript.
  }
}
```

```
<s:Button>

  <fx:Script>

    // Override inherited methods
    // and properties.

    // Define new methods
    // and properties.

    // Define custom logic
    // in ActionScript.

  </fx:Script>

  <!-- Add MXML code. -->

</s:Button>
```

Both implementations create a component as a subclass of the Button class and, therefore, inherit all of the public and protected properties, methods, and other elements of the Button class. Within each implementation, you can override inherited items, define new items, and add your custom logic.

***Note:** You cannot override an inherited property defined by a variable, but you can override a property defined by setter and getter methods. You can reset the value of an inherited property defined by a variable. You typically reset it in the constructor of the subclass for an ActionScript component, or in an event handler for an MXML component because MXML components cannot define a constructor.*

However, when you use MXML, the Flex compiler performs most of the overhead required to create a subclass of a component for you. This makes it much easier to create components in MXML than in ActionScript.

Deciding to create components in MXML or ActionScript

One of the first decisions that you must make when creating custom components is deciding whether to write them in MXML or in ActionScript. Ultimately, it is the requirements of your application that determine how you develop your custom component.

Some basic guidelines include the following:

- MXML components and ActionScript components both define new ActionScript classes.
- Almost anything that you can do in a custom ActionScript custom component, you can also do in a custom MXML component. However, for simple components, such as components that modify the behavior of an existing component or add a basic feature to an existing component, it is simpler and faster to create them in MXML.
- When your new component is a composite component that contains other components, and you can express the positions and sizes of those other components using one of the Flex containers, you should use MXML to define your component.
- To modify the behavior of the component, such as the way a container lays out its children, use ActionScript.
- To create a visual component by creating a subclass from UIComponent, use ActionScript.
- To create a nonvisual component, such as a formatter, validator, or effect, use ActionScript.
- To add logging support to your control, use ActionScript. For more information, see “Logging” on page 2222.

***Note:** The Adobe® Flash® Professional 8 authoring environment does not support ActionScript 3.0. Therefore, you should not use it to create ActionScript components for Flex. Instead, you should use Adobe® Flash® Builder™ or Adobe Flash Professional CS4 or later.*

For more information on custom MXML components, see “Simple MXML components” on page 2399. For more information on ActionScript components, see “Create simple visual components in ActionScript” on page 2433.

Creating new components

Your application might require you to create new components, rather than modifying existing ones. To create components, you typically create them in ActionScript by creating a subclass from the UIComponent class. This class contains the generic functionality of all Flex components. You then add the required functionality to your new component to meet your application requirements.

For more information, see “Create advanced MX visual components in ActionScript” on page 2475.

Creating custom components

You create custom components as either MXML or ActionScript files.

Creating MXML components

Flex supplies the Spark ComboBox control that you can use as part of a form that collects address information from a customer. In the form, you can include a ComboBox control to let the user select the state portion of the address from a list of the 50 states in the U.S. In an application that has multiple forms where a user can enter an address, it would be tedious to create and initialize multiple ComboBox controls with the same information about all 50 states.

Instead, you create an MXML component that contains a ComboBox control with all the 50 states defined within in it. Then, wherever you need to add a state selector to your application, you use your custom MXML component. The following example shows a possible definition for a custom ComboBox control:

```
<?xml version="1.0"?>
<!-- createcomps_intro\StateComboBox.mxml -->
<!-- Specify the root tag and namespace. -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
            <!-- Add all other states. -->
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

This example shows the following:

1. The first line of the custom MXML component definition specifies the declaration of the XML version.
2. The first MXML tag of the component, called its *root* tag, specifies a Flex component or a custom component. MXML components correspond to ActionScript classes, therefore, the root tag specifies the superclass of the MXML component. In this example, the MXML component specifies the Flex ComboBox control as its superclass.
3. The `xmlns` properties in the root tag specifies the Flex XML namespaces. In this example, the `xmlns` property indicates that tags in the MX namespace use the prefix *mx:*.
4. The remaining lines of the component specify its definition.

The main application, or any other MXML component file, references the StateComboBox component, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_intro/IntroMyApplication.mxml -->
<!-- Include the namespace definition for your custom components. -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">
    <!-- Use the filename as the MXML tag name. -->
    <MyComp:StateComboBox/>
</s:Application>
```

The MXML tag name for a custom component is composed of two parts: the namespace prefix, in this case `MyComp`, and the tag name. The namespace prefix tells Flex where to look for the file that implements the custom component. The tag name corresponds to the filename of the component, in this case StateComboBox.mxml. Therefore, a file named StateComboBox.mxml defines a component with the tag name of `<`*namespace*`:StateComboBox>`.

As part of the `<s:Application>` tag, the main application file includes the following namespace definition: `xmlns:MyComp="*"`. This definition specifies that the component is in the same directory as the main application file, or in a directory included in the ActionScript source path. For more information on deploying MXML components, see "Simple MXML components" on page 2399.

The best practice is to put your custom components in a subdirectory of your application. That practice helps to ensure that you do not have duplicate component names because they have a different namespace. If you stored your component in the myComponents subdirectory of your application, you would specify the namespace definition as `xmlns:MyComp="myComponents.*"`.

The StateComboBox.mxml file specifies the ComboBox control as its root tag, so you can reference all of the properties of the ComboBox control in the MXML tag of your custom component, or in the ActionScript specified in an `<fx:Script>` tag. For example, the following example specifies the `ComboBox.maxChars` property and a listener for the `ComboBox.close` event for your custom control:

```
<?xml version="1.0"?>
<!-- createcomps_intro/MyApplicationProperties.mxml -->
<s:Application  xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            private function handleCloseEvent(eventObj:Event):void {
                // ...
            }
        ]]>
    </fx:Script>
    <MyComp:StateComboBox maxChars="25"
        close="handleCloseEvent(event);"/>
</s:Application>
```

For more information on MXML components, see "Simple MXML components" on page 2399.

Creating ActionScript components

You create ActionScript components by defining ActionScript classes. You can create the following types of components in ActionScript:

User-interface, or visual, components User-interface components contain both processing logic and visual elements. You create custom user-interface components to modify existing behavior or add new functionality to the component. These components usually extend the Flex component hierarchy. You can extend from the UIComponent class, or any of the Flex components, such as Button, ComboBox, or DataGrid. Your custom ActionScript component inherits all of the methods, properties, events, styles, and effects of its superclass.

Nonvisual components Nonvisual components define nonvisual elements. Flex includes several types of nonvisual components that you can create, including formatters, validators, and effects. You create nonvisual components by creating a subclass from the Flex component hierarchy. For validators, you create subclasses of the Validator class; for formatters you create subclasses of the Formatter class; and for effects, you create subclasses of the Effect class.

For example, you can define a custom button component based on the Spark Button class, as the following example shows:

```
package myComponents
{
    // createcomps_intro/myComponents/MyButton.as
    import spark.components.Button;
    public class MyButton extends Button {

        // Define the constructor.
        public function MyButton() {
            // Call the constructor in the superclass.
            super();
            // Set the label property to "Submit".
            label="Submit";
        }
    }
}
```

In this example, you write your MyButton class to the MyButton.as file.

You must define your custom components within an ActionScript package. The package reflects the directory location of your component in the directory structure of your application. Typically, you put custom ActionScript components in directories that are in the ActionScript source path, subdirectories of your application, or for Adobe® LiveCycle™ Data Services ES, in the WEB-INF/flex/user_classes directory. In this example, the `package` statement specifies that the MyButton.as file is in the myComponents subdirectory of your Flex application.

In the MXML file that references the custom component, you define the namespace and reference it in an MXML file as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_intro/MyApplicationASComponent.mxml -->
<!-- Include the namespace definition for your custom components. -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <!-- Use the filename as the MXML tag name. -->
    <MyComp:MyButton/>
</s:Application>
```

In this example, you first define the `MyComp` namespace that specifies the location of your custom component in the application's directory structure. You then reference the component as an MXML tag using the namespace prefix.

For more information, see "Create simple visual components in ActionScript" on page 2433.

Deploying components

When you deploy your custom components as MXML or ActionScript files, you typically deploy them in the same directory structure as your application files, in a directory specified in the ActionScript source path, or for LiveCycle Data Services ES, in the WEB-INF/flex/user_classes directory.

For security reasons, you may decide not to deploy your custom components as source code files. Alternatively, you can deploy your components as SWC files or as part of a Runtime Shared Library (RSL).

A *SWC file* is an archive file for Flex components. SWC files make it easy to exchange components among Flex developers. You need only exchange a single file, rather than the MXML or ActionScript files and images and other resource files. In addition, the SWF file inside a SWC file is compiled, which means that the code is hidden from casual view.

SWC files can contain one or more components and are packaged and expanded with the PKZip archive format. You can open and examine a SWC file using WinZip, JAR, or another archiving tool. However, you should not manually change the contents of a SWC file, and you should not try to run the SWF file that is in a SWC file outside of a SWC file.

To create a SWC file, use the compc utility in the *flex_install_dir*/bin directory. The compc utility generates a SWC file from MXML component source files and/or ActionScript component source files. For more information on compc, see "Flex compilers" on page 2164.

One way to reduce the size of your application's SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but only need to be transferred to the client once. These shared files are known as Runtime Shared Libraries or RSLs.

For more information, including information on how to create an RSL file, see "Runtime Shared Libraries" on page 253.

Custom ActionScript components

You use ActionScript code to create ActionScript components for Adobe® Flex®, or to add logic to MXML components. ActionScript provides flow control and object manipulation features that are not available in MXML.

The summary of the general rules for using ActionScript code in custom components in this topic supplements the information in the ActionScript reference documentation. For additional information on ActionScript, see the following resources:

- *ActionScript 3.0 Reference for the Adobe Flash Platform*: Contains the API reference for ActionScript 3.0.
- *ActionScript 3.0 Developer's Guide*: Contains information on using ActionScript 3.0.

Using ActionScript

Before you start developing custom components, you should be familiar with basic ActionScript coding practices.

Using the package statement

You must define your ActionScript custom components within a package. The package reflects the directory location of your component within the directory structure of your application. To define the package structure, you include the package statement in your class definition, as the following example shows:

```
package myComponents
{
    // Class definition goes here.
}
```

Your `package` statement must wrap the entire class definition. If you write your ActionScript class file to the same directory as your other application files, you can leave the package name blank. However, as a best practice, you should store your components in a subdirectory, where the package name reflects the directory location. In this example, write your ActionScript class file to the directory myComponents, a subdirectory of your main application directory.

Formatters are a particular type of component. You might also create a subdirectory of your application's root directory called myFormatters for all of your custom formatter classes. Each formatter class would then define its `package` statement, as the following example shows:

```
package myFormatters
{
    // Formatter class definition goes here.
}
```

If you create a component that is shared among multiple applications, or a component that might be used with third-party components, assign a unique package name to avoid naming conflicts. For example, you might prefix your package name with your company name, as in:

```
package Acme.myFormatters
{
    // Formatter class definition goes here.
}
```

When you reference a custom component from an MXML file, specify a namespace definition for the component that corresponds to its directory location and package name, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myFormatters.*">

    <!-- Declare a formatter and specify formatting properties. -->
    <MyComp:SimpleFormatter id="upperFormat" formatString="upper"/>
        ...

</s:Application>
```

If a formatter class is in a subdirectory of myFormatters, such as myFormatters/dataFormatters, the package statement is as follows:

```
package myFormatters.dataFormatters
{
    // Formatter class definition goes here.
}
```

You then specify the namespace definition for the component, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myFormatters.dataFormatters.*">

    <!-- Declare a formatter and specify formatting properties. -->
    <MyComp:SimpleFormatter id="upperFormat" formatString="upper"/>
        ...

</s:Application>
```

Using the import statement

You use the import statement to import any classes that your class requires. Importing adds a reference to the class so that you can access classes defined by the import. Classes that you import must be located in the ActionScript source path for your application.

You import the classes referenced by your custom component as part of its implementation, as the following example shows:

```
package myComponents
{
    // Import necessary classes.
    import mx.core.Container;
    import mx.controls.Button;
    // Import all classes in the mx.events package
    import mx.events.*;

    // Class definition goes here.

    // You can now create an instance of a Container using this syntax:
    private var myContainer:Container = new Container();

}
```

There is a distinct difference between including and importing in ActionScript. *Including* is copying lines of code from one ActionScript file into another. Files that you include must be located relative to the file performing the include, or use an absolute path. *Importing* is adding a reference to a class file or package so that you can access objects and properties defined by external classes.

For more information on including and importing, see "Using ActionScript" on page 32.

Using the class statement

You use the class statement to define your class name, and to specify its superclass, as the following example shows:

```
package myComponents
{
    // Import necessary classes
    import mx.core.Container;
    import mx.controls.Button;
    // Import all classes in the mx.events package
    import mx.events.*;

    // Class definition goes here.
    public class MyButton extends Button {

        // Define properties, constructor, and methods.

    }
}
```

The class definition of your component must be prefixed by the `public` keyword, or it cannot be used as an MXML tag. A file that contains a class definition can have one, and only one, public class definition, although it can have additional internal class definitions. Place any internal class definitions at the bottom of your source file below the closing curly brace of the package definition.

In a single ActionScript file, you can define only one class in the package. To define more than one class in a file, define the additional classes outside of the package body.

***Note:** The class definition is one of the few ActionScript constructs that you cannot use in an `<fx:Script>` block in an MXML file.*

Defining the constructor

An ActionScript class must define a public constructor method, which initializes an instance of the class. The constructor has the following characteristics:

- No return type.
- Should be declared public.
- Might have optional arguments.
- Cannot have any required arguments if you use it as an MXML tag.
- Calls the `super()` method to invoke the superclass' constructor.

You call the super() method within your constructor to invoke the superclass' constructor to initialize the inherited items from the superclass. The `super()` method should be the first statement in your constructor; otherwise, the inherited parts of the superclass might not be properly constructed. In some cases, you might want to initialize your class first, and then call `super()`.

***Note:** If you do not define a constructor, the compiler inserts one for you and adds a call to `super()`. However, it is considered a best practice to write a constructor and to explicitly call `super()`, unless the class contains nothing but static members. If you define the constructor, but omit the call to `super()`, Flex automatically calls `super()` at the beginning of your constructor.*

In the following example, you define a constructor that uses `super()` to call the superclass' constructor:

```
package myComponents
{
    // Import necessary classes
    import mx.core.Container;
    import mx.controls.Button;
    // Import all classes in the mx.events package
    import mx.events.*;

    // Class definition goes here.
    public class MyButton extends Button {

        // Public constructor.
        public function MyButton()
        {
            // Call the constructor in the superclass.
            super();
        }
        // Define properties and methods.

    }
}
```

***Note:** You cannot define a constructor for an MXML component. For more information, see "About implementing IMXMLObject" on page 2431*

Defining properties as variables

Properties let you define data storage within your class. You can define your properties as public, which means that they can be accessed by users of the class. You can also define properties as private, which means that they are used internally by the class, as the following example shows:

```
public class MyButton extends Button {

    // Define private vars.
    private var currentFontSize:Number;

    // Define public vars.
    public var maxFontSize:Number = 15;
    public var minFontSize:Number = 5;
}
```

Users of the class can access the public variables but not the private variables, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myControls.*">

    <MyComp:MyButton label="Submit" maxFontSize="30"/>
</s:Application>
```

Although you can define your classes to use public properties, you may find it advantageous to define properties by using setter and getter methods. For more information, see "Defining methods" on page 2368.

***Note:** You cannot override an inherited property defined by a variable, but you can override a property defined by setter and getter methods. You can reset the value of an inherited property defined by a variable. You typically reset it in the constructor of the subclass for an ActionScript component, or in an event handler for an MXML component because MXML components cannot define a constructor.*

Defining properties as getters and setters

You can define properties for your components by using setter and getter methods. The advantage of getters and setters is that they isolate the variable from direct public access so that you can perform the following actions:

- Inspect and validate any data written to the property on a write
- Trigger events that are associated with the property when the property changes
- Calculate a return value on a read
- Allow a child class to override

To define getter and setter methods, precede the method name with the keyword get or set, followed by a space and the property name. The following example shows the declaration of a public property named `initialCount`, and the getter and setter methods that get and set the value of this property:

```
// Define internal private variable.
private var _initialCount:uint = 42;

// Define public getter.
public function get initialCount():uint {
    return _initialCount;
}

// Define public setter.
public function set initialCount(value:uint):void {
    _initialCount = value;
}
```

By convention, setters use the identifier `value` for the name of the argument.

The variable that stores the property's value cannot have the same name as the getter or setter. By convention, precede the name of the variables with one (_) or two underscores (__). In addition, Adobe recommends that you declare the variable as private or protected.

Users of the class can access the public property, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myControls.*" >

    <MyComp:MyButton label="Submit" initialCount="24"/>
</s:Application>
```

If the getter or setter overrides a getter or setter in a superclass, ensure that you include the `override` keyword, as the following example shows:

```
override public function get label():String {}
override public function set label(value:String):void {}
```

Defining methods

Methods define the operations that your class can perform. You define methods in the body of the class. Your methods can override a method of a superclass, or define new functionality for your components.

If the method adds new functionality, you define it using the `function` keyword, as the following example shows:

```
public function myMethod():void {
    // Method definition
}
```

If you define this method as a public method, users of the class can call it.

You can also define private methods, as the following example shows:

```
private function internalMethod():void {
    // Method definition
}
```

Private methods are for internal use by the class, and cannot be called by users of the class.

If the method overrides a method in a superclass, you must include the `override` keyword and the signature of the method must exactly match that of the superclass method, as the following example shows:

```
override protected function createChildren():void {
    // Method definition
}
```

Your methods may take required or optional arguments. To make any of the arguments optional, assign default values to them, as the following example shows:

```
override public validate(value:Object = null,
    supressEvents:Boolean = false):ValidationResultEvent {
    // Method definition
}
```

If the method takes a variable number of arguments, use the "..." syntax, as the following example shows:

```
function foo(n:Number, ... rest):void {
    // Method definition
}
```

Flex creates an Array called `rest` for the optional arguments. Therefore, you can determine the number of arguments passed to the method by using `rest.length`, and access the arguments by using `rest[i]`.

Using the super keyword in a method override

You use the super keyword in a method override to invoke the corresponding method of the superclass. The `super` keyword has the following syntax:

```
super.methodName([arg1, ..., argN])
```

This technique is useful when you create a subclass method that adds behavior to a superclass method but also invokes the superclass method to perform its original behavior.

***Note:** Although Flex automatically calls the `super()` method in a constructor to execute the superclass' constructor, you must call `super.methodName()` in a method override. Otherwise, the superclass' version of the method does not execute.*

Whether you call `super.`*myMethod*`()` within a method override depends on your application requirement, as follows:

- Typically, you extend the existing functionality of the superclass method, so the most common pattern is to call `super.`*myMethod*`()` first in your method override, and then add your logic.
- You might need to change something before the superclass method does its work. In this case, you might call `super.`*myMethod*`()` in the override after your logic.
- In some method overrides, you might not want to invoke the superclass method at all. Only call `super.`*myMethod*`()` if and when you want the superclass to do its work.
- Sometimes the superclass has an empty method that does nothing, which requires you to implement the functionality in the method. In this case, you should still call `super.`*myMethod*`()` because in a future version of Flex, that method might implement some functionality. For more information, see the documentation on each Flex class.

About the scope

Scoping is mostly a description of what the `this` keyword refers to at any given point in your application. In the main MXML application file, the file that contains the `<s:Application>` tag, the current scope is the Application object, and therefore the `this` keyword refers to the Application object.

In an ActionScript component, the scope is the component itself and not the application or other file that references the component. As a result, the `this` keyword inside the component refers to the component instance and not the Flex Application object.

Nonvisual ActionScript components do not have access to their parent application with the `parentDocument` property. However, you can access the top-level Application object by using the `mx.core.Application.application` property.

For more information on scope, see "Using ActionScript" on page 32.

Custom events

You can create custom events as part of defining MXML and ActionScript components. Custom events let you add functionality to your custom components to respond to user interactions, to trigger actions by your custom component, and to take advantage of data binding.

For more information on creating custom events for MXML components, see “Advanced MXML components” on page 2412. For information on creating custom events for ActionScript components, see “Create simple visual components in ActionScript” on page 2433.

About events

Adobe Flex applications are event-driven. Events let an application know when the user interacts with the interface, and also when important changes happen in the appearance or life cycle of a component, such as the creation of a component or its resizing. Events can be generated by user input devices, such as the mouse and keyboard, or by the asynchronous operations, such as the return of a web service call or the firing of a timer.

The core class of the Flex component architecture, mx.core.UIComponent, defines core events, such as `updateComplete`, `resize`, `move`, `creationComplete`, and others that are fundamental to all components. Subclasses of UIComponent inherit these events.

Custom components that extend existing Flex classes inherit all the events of the base class. Therefore, if you extend the Button class to create the MyButton class, you can use the `click` event and the events that all controls inherit, such as `mouseOver` or `initialize`, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_events/EventsMyApplication.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            // Event listener for the click event.
            private function handleClick(eventObj:Event):void {
                // Define event listener.
            }
            // Event listener for the initialize event.
            private function handleInit(eventObj:Event):void {
                // Define event listener.
            }
        ]]>
    </fx:Script>
    <MyComp:MyButton
        click="handleClick(event);"
        initialize="handleInit(event);"/>
</s:Application>
```

In addition to using the events inherited from its superclasses, your custom components can define custom events. You use custom events to support data binding, to respond to user interactions, or to trigger actions by your component.

For more information on the Flex event mechanism, see “Events” on page 54.

Using an event object

When a Flex component dispatches an event, it creates an event object, where the properties of the event object contain information describing the event. An event listener takes this event object as an argument and accesses the properties of the object to determine information about the event.

The base class for all event objects is the flash.events.Event class. All event objects are instances of the Event class, or instances of a subclass of the Event class.

The following table describes the public properties of the Event class. The Event class implements these properties using getter methods.

Property	**Type**	**Description**
`type`	String	The name of the event; for example, `"click"` . The event constructor sets this property.
`target`	EventDispatcher	A reference to the component instance that dispatches the event. This property is set by the `dispatchEvent()` method; you cannot change this to a different object.
`currentTarget`	EventDispatcher	A reference to the component instance that is actively processing the Event object. The value of this property is different from the value of the `target` property during the event capture and bubbling phase. For more information, see "Events" on page 54.
`eventPhase`	uint	The current phase in the event flow. The property might contain the following values: • `EventPhase.CAPTURING_PHASE`: The capture phase • `EventPhase.AT_TARGET`: The target phase • `EventPhase.BUBBLING_PHASE`: The bubbling phase
`bubbles`	Boolean	Whether an event is a bubbling event. If the event can bubble, the value for this property is `true;` otherwise, it is `false`. You can optionally pass this property as a constructor argument to the Event class. By default, most event classes set this property to `false`. For more information, see "Events" on page 54.
`cancelable`	Boolean	Whether the event can be canceled. If the event can be canceled, the value for this value is `true;` otherwise, it is `false`. You can optionally pass this property as a constructor argument to the Event class. By default, most event classes set this property to `false`. For more information, see "Events" on page 54.

Dispatching custom events

Flex defines many of the most common events, such as the `click` event for the Button control; however, your application may require that you create events. In your custom Flex components, you can dispatch any of the predefined events inherited by the component from its superclass, and dispatch new events that you define within the component.

To dispatch a new event from your custom component, you must do the following:

1. (Optional) Create a subclass from the flash.events.Event class to create an event class that describes the event object. For more information, see "Creating a subclass from the Event class" on page 2371.
2. (Optional) Use the `[Event]` metadata tag to make the event public so that the MXML compiler recognizes it. For more information, see "Using the Event metadata tag" on page 2373.
3. Dispatch the event using the `dispatchEvent()` method. For more information, see "Dispatching an event" on page 2374.

Creating a subclass from the Event class

All events use an event object to transmit information about the event to the event listener, where the base class for all event objects is the flash.events.Event class. When you define a custom event, you can dispatch an event object of the Event type, or you can create a subclass of the Event class to dispatch an event object of a different type. You typically create a subclass of the Event class when your event requires you to add information to the event object, such as a new property to hold information that the event listener requires.

For example, the event objects associated with the Flex Tree control include a property named `node`, which identifies the node of the Tree control associated with the event. To support the `node` property, the Tree control dispatches event objects of type TreeEvent, a subclass of the Event class.

Within your subclass of the Event class, you can add properties, add methods, set the value of an inherited property, or override methods inherited from the Event class. For example, you might want to set the `bubbles` property to `true` to override the default setting of `false`, which is inherited from the Event class.

You are required to override the `Event.clone()` method in your subclass. The `clone()` method returns a cloned copy of the event object by setting the `type` property and any new properties in the clone. Typically, you define the `clone()` method to return an event instance created with the `new` operator.

Suppose that you want to pass information about the state of your component to the event listener as part of the event object. To do so, you create a subclass of the Event class to create an event, EnableChangeEvent, as the following example shows:

```
package myEvents
{
    //createcomps_events/myEvents/EnableChangeEvent.as
    import flash.events.Event;
    public class EnableChangeEvent extends Event
    {
        // Public constructor.
        public function EnableChangeEvent(type:String,
            isEnabled:Boolean=false) {
                // Call the constructor of the superclass.
                super(type);

                // Set the new property.
                this.isEnabled = isEnabled;
        }
        // Define static constant.
        public static const ENABLE_CHANGE:String = "enableChange";
        // Define a public variable to hold the state of the enable property.
        public var isEnabled:Boolean;
        // Override the inherited clone() method.
        override public function clone():Event {
            return new EnableChangeEvent(type, isEnabled);
        }
    }
}
```

In this example, your custom class defines a public constructor that takes two arguments:

- A String value that contains the value of the `type` property of the Event object.
- An optional Boolean value that contains the state of the component’s `isEnabled` property. By convention, all constructor arguments for class properties, except for the `type` argument, are optional.

  From within the body of your constructor, you call the `super()` method to initialize the base class properties.

Using the Event metadata tag

You use the `[Event]` metadata tag to define events dispatched by a component so that the Flex compiler can recognize them as MXML tag attributes in an MXML file. You add the `[Event]` metadata tag in one of the following locations:

- ActionScript components

  Above the class definition, but within the package definition, so that the events are bound to the class and not a particular member of the class.

- MXML components

  In the `<fx:Metadata>` tag of an MXML file.

The `Event` metadata keyword has the following syntax:

```
[Event(name="eventName", type="package.eventType")]
```

The *eventName* argument specifies the name of the event. The *eventType* argument specifies the class name, including the package, that defines the event object.

The following example identifies the `enableChange` event as an event that an ActionScript component can dispatch:

```
[Event(name="enableChange", type="myEvents.EnableChangeEvent")]
public class MyComponent extends TextArea
{
    ...
}
```

The following example shows the `[Event]` metadata tag within the `<fx:Metadata>` tag of an MXML file:

```
<?xml version="1.0"?>
<!-- createcomps_events\myComponents\MyButton.mxml -->
<s:Button xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    click="dispatchEvent(new EnableChangeEvent('enableChange'));">
    <fx:Script>
        <![CDATA[
            import myEvents.EnableChangeEvent;
        ]]>
    </fx:Script>
    <fx:Metadata>
        [Event(name="enableChange", type="myEvents.EnableChangeEvent")]
    </fx:Metadata>
</s:Button>
```

Once you define the event using the `[Event]` metadata tag, you can refer to the event in an MXML file, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_events/MainEventApp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import myEvents.EnableChangeEvent;
            public function
                enableChangeListener(eventObj:EnableChangeEvent):void {
                    // Handle event.
                    myTA2.text='got event';
            }
        ]]>
    </fx:Script>
    <MyComp:MyButton enableChange="myTA.text='got event';" />
    <mx:TextArea id="myTA" />
    <MyComp:MyButton enableChange="enableChangeListener(event);" />
    <mx:TextArea id="myTA2" />
</s:Application>
```

If you do not identify an event with the `[Event]` metadata tag, the compiler generates an error if you try to use the event name in MXML. The metadata for events is inherited from the superclass, however, so you do not need to tag events that are already defined with the `[Event]` metadata tag in the superclass.

Dispatching an event

You use the dispatchEvent() method to dispatch an event. The `dispatchEvent()` method has the following signature:

```
public dispatchEvent(event:Event):Boolean
```

This method requires an argument of the Event type, which is the event object. The `dispatchEvent()` method initializes the `target` property of the event object with a reference to the component dispatching the event.

You can create an event object and dispatch the event in a single statement, as the following example shows:

```
dispatchEvent(new Event("click"));
```

You can also create an event object, initialize it, and then dispatch it, as the following example shows:

```
var eventObj:EnableChangeEvent = new EnableChangeEvent("enableChange");
eventObj.isEnabled=true;
dispatchEvent(eventObj);
```

For complete examples that create and dispatch custom events, see “Advanced MXML components” on page 2412 and “Create simple visual components in ActionScript” on page 2433.

Creating static constants for the Event.type property

The constructor of an event class typically takes a single required argument that specifies the value of the event object’s `type` property. In the previous section, you passed the string `enableChange` to the constructor, as the following example shows:

```
// Define event object, initialize it, then dispatch it.
var eventObj:EnableChangeEvent = new EnableChangeEvent("enableChange");
dispatchEvent(eventObj);
```

The Flex compiler does not examine the string passed to the constructor to determine if it is valid. Therefore, the following code compiles, even though `enableChangeAgain` might not be a valid value for the `type` property:

```
var eventObj:EnableChangeEvent =
    new EnableChangeEvent("enableChangeAgain");
```

Because the compiler does not check the value of the `type` property, the only time that your application can determine if `enableChangeAgain` is valid is at run time.

However, to ensure that the value of the `type` property is valid at compile time, Flex event classes define static constants for the possible values for the `type` property. For example, the Flex EffectEvent class defines the following static constant:

```
// Define static constant for event type.
public static const EFFECT_END:String = "effectEnd";
```

To create an instance of an EffectEvent class, you use the following constructor:

```
var eventObj:EffectEvent = new EffectEvent(EffectEvent.EFFECT_END);
```

If you incorrectly reference the constant in the constructor, the compiler generates a syntax error because it cannot locate the associated constant. For example, the following constructor generates a syntax error at compile time because `MY_EFFECT_END` is not a predefined constant of the EffectEvent class:

```
var eventObj:EffectEvent = new EffectEvent(EffectEvent.MY_EFFECT_END);
```

You can use this technique when you define your event classes. The following example modifies the definition of the EnableChangeEventConst class to include a static constant for the `type` property:

```
package myEvents
{
    //createcomps_events/myEvents/EnableChangeEventConst.as
    import flash.events.Event;
    public class EnableChangeEventConst extends Event
    {
        // Public constructor.
        public function EnableChangeEventConst(type:String,
            isEnabled:Boolean=false) {
                // Call the constructor of the superclass.
                super(type);

                // Set the new property.
                this.isEnabled = isEnabled;
        }
        // Define static constant.
        public static const ENABLE_CHANGE:String = "myEnable";

        // Define a public variable to hold the state of the enable property.
        public var isEnabled:Boolean;
        // Override the inherited clone() method.
        override public function clone():Event {
            return new EnableChangeEvent(type, isEnabled);
        }
    }
}
```

Now you create an instance of the class by using the static constant, as the following example shows for the MyButtonConst custom component:

```
<?xml version="1.0"?>
<!-- createcomps_events\myComponents\MyButtonConst.mxml -->
<s:Button xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    click="dispatchEvent(new EnableChangeEventConst(EnableChangeEventConst.ENABLE_CHANGE));">
    <fx:Script>
        <![CDATA[
            import myEvents.EnableChangeEventConst;
        ]]>
    </fx:Script>
    <fx:Metadata>
        [Event(name="myEnable", type="myEvents.EnableChangeEventConst")]
    </fx:Metadata>
</s:Button>
```

This technique does not preclude you from passing a string to the constructor.

Metadata tags in custom components

You insert metadata tags into your MXML and ActionScript files to provide information to the Adobe® Flex® compiler. Metadata tags do not get compiled into executable code, but provide information to control how portions of your code get compiled.

For more information about additional metadata tags that you use when creating an application, such as the `[Embed]` metadata tag, see "Embedding assets" on page 1699.

About metadata tags

Metadata tags provide information to the compiler that describes how your components are used in an application. For example, you might create a component that defines a new event. To make that event known to the Flex compiler so that you can reference it in MXML, you insert the `[Event]` metadata tag into your component, as the following ActionScript class definition shows:

```
[Event(name="enableChanged", type="flash.events.Event")]
class ModalText extends TextArea {
    ...
}
```

In this example, the `[Event]` metadata tag specifies the event name and the class that defines the type of the event object dispatched by the event. After you identify the event to the compiler, you can reference it in MXML, as the following example shows:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" xmlns:MyComp="*">

    <fx:Script>
        <![CDATA[
            function handleEnableChangeEvent(eventObj:Event):void {
                ...
            }
        ]]>
    </fx:Script>

    <MyComp:ModalText enableChanged="handleEnableChangeEvent(event);"/>
</s:Application>
```

If you omit the `[Event]` metadata tag from your class definition, Flex issues a syntax error when it compiles your MXML file. The error message indicates that Flex does not recognize the `enableChanged` property.

The Flex compiler recognizes component metadata statements in your ActionScript class files and MXML files. The metadata tags define component attributes, data binding properties, events, and other properties of the component. Flex interprets these statements during compilation; they are never interpreted during run time.

Metadata statements are associated with a class declaration, an individual data field, or a method. They are bound to the next line in the file. When you define a component property or method, add the metadata tag on the line before the property or method declaration.

Metadata tags in ActionScript

In an ActionScript file, when you define component events or other aspects of a component that affect more than a single property, you add the metadata tag outside the class definition so that the metadata is bound to the entire class, as the following example shows:

```
// Add the [Event] metadata tag outside of the class file.
[Event(name="enableChange", type="flash.events.Event")]
public class ModalText extends TextArea {

    ...

    // Define class properties/methods
    private var _enableTA:Boolean;

    // Add the [Inspectable] metadata tag before the individual property.
    [Inspectable(defaultValue="false")]
    public function set enableTA(val:Boolean):void {
        _enableTA = val;
        this.enabled = val;

        // Define event object, initialize it, then dispatch it.
        var eventObj:Event = new Event("enableChange");
        dispatchEvent(eventObj);
    }
}
```

In this example, you add the [Event] metadata tag before the class definition to indicate that the class dispatches an event named enableChanged. You also include the [Inspectable] metadata tag to indicate the default value of the property for Adobe® Flash® Builder™. For more information on using this tag, see “Inspectable metadata tag” on page 2386.

Metadata tags in MXML

In an MXML file, you insert the metadata tags either in an <fx:Script> block along with your ActionScript code, or in an <fx:Metadata> block, as the following example shows:

```
<?xml version="1.0"?>
<!-- TextAreaEnabled.mxml -->
<mx:TextArea xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Metadata>
        [Event(name="enableChange", type="flash.events.Event")]
    </fx:Metadata>

    <fx:Script>
        <![CDATA[

            // Import Event class.
            import flash.events.Event;

            // Define class properties and methods.
            private var _enableTA:Boolean;

            // Add the [Inspectable] metadata tag before the individual property.
            [Inspectable(defaultValue="false")]
            public function set enableTA(val:Boolean):void {
                _enableTA = val;
                this.enabled = val;

                // Define event object, initialize it, then dispatch it.
                var eventObj:Event = new Event("enableChange");
                dispatchEvent(eventObj);
            }
        ]]>
    </fx:Script>
</mx:TextArea>
```

A key difference between the <fx:Metadata> and <fx:Script> tags is that text within the <fx:Metadata> tag is inserted before the generated class declaration, but text within <fx:Script> tag is inserted in the body of the generated class declaration. Therefore, metadata tags like [Event] and [Effect] must go in an <fx:Metadata> tag, but the [Bindable] and [Embed] metadata tags must go in an <fx:Script> tag.

Metadata tags

The following table describes the metadata tags that you can use in ActionScript class files:

Tag	Description
`[Alternative]`	Specifies a replacement class for an existing class, and a version number that indicates when the replacement occurred. For more information, see "Alternative metadata tag" on page 2380.
`[ArrayElementType]`	Defines the allowed data type of each element of an Array. For more information, see "ArrayElementType metadata tag" on page 2380.
`[Bindable]`	Identifies a property that you can use as the source of a data binding expression. For more information, see "Bindable metadata tag" on page 2381.
`[DefaultProperty]`	Defines the name of the default property of the component when you use the component in an MXML file. For more information, see "DefaultProperty metadata tag" on page 2383.
`[Deprecated]`	Marks a class or class element as deprecated so that the compiler can recognize it and issue a warning when the element is used in an application. For more information, see "Deprecated metadata tag" on page 2383.
`[Effect]`	Defines the MXML property name for the effect. For more information, see "Effect metadata tag" on page 2384.
`[Embed]`	Imports JPEG, GIF, PNG, SVG, and SWF files at compile time. Also imports image assets from SWC files. This is functionally equivalent to the MXML `@Embed` syntax, as described in "Embedding assets" on page 1699.
`[Event]`	Defines the MXML property for an event and the data type of the event object that a component emits. For more information, see "Event metadata tag" on page 2384.
`[Exclude]`	Omits an inherited class element from the Flash Builder tag inspector. The syntax is as follows: `[Exclude(name="label", kind="property")]` Where `kind` can be `property`, `method`, `event`, or `style`.
`[ExcludeClass]`	Omits the class from the Flash Builder tag inspector. This is equivalent to the `@private` tag in ASDoc when applied to a class.
`[HostComponent]`	Specifies the host component for a Spark skin class. For more information, see "HostComponent metadata tag" on page 2385
`[IconFile]`	Identifies the filename for the icon that represents the component in the Insert bar of Adobe Flash Builder. For more information, see "IconFile metadata tag" on page 2386.
`[Inspectable]`	Defines an attribute exposed to component users in the attribute hints and Tag inspector of Flash Builder. Also limits allowable values of the property. For more information, see "Inspectable metadata tag" on page 2386.
`[InstanceType]`	Specifies the allowed data type of a property of type IDeferredInstance. For more information, see "InstanceType metadata tag" on page 2387.
`[NonCommittingChangeEvent]`	Identifies an event as an interim trigger. For more information, see "NonCommittingChangeEvent metadata tag" on page 2388.
`[RemoteClass]`	Maps the ActionScript object to a Java object. For more information on using the `[RemoteClass]` metadata tag, see "RemoteClass metadata tag" on page 2388.
`[RichTextContent]`	Indicate that the value of a property in MXML should always be interpreted by the compiler as a String. For more information, see "RichTextContent metadata tag" on page 2389.

Tag	Description
`[SkinPart]`	Define a property of a component that corresponds to a skin part. For more information, see "SkinPart metadata tag" on page 2389.
`[SkinState]`	Defines the view states that a component's skin must support. For more information, see "SkinState metadata tag" on page 2390.
`[Style]`	Defines the MXML property for a style property for the component. For more information on using the `[Style]` metadata tag, see "Style metadata tag" on page 2390.
`[SWF]`	Specifies attributes of the application when you write the main application file in ActionScript. For more information, see "SWF metadata tag" on page 2392.
`[Transient]`	Identifies a property that should be omitted from data that is sent to the server when an ActionScript object is mapped to a Java object using `[RemoteClass]`. For more information, see "Transient metadata tag" on page 2392.

Alternative metadata tag

If you want to replace one class with another, mark the class to be replaced with the `[Alternative]` metadata tag. The `[Alternative]` metadata tag specifies the replacement class, and a version number that indicates when the replacement occurred.

The `[Alternative]` metadata tag is not the same a the `[Deprecated]` metadata tag. When a class is deprecated, it might not work in a future release. A class marked by the `[Alternative]` metadata tag is still supported, but it indicates that there is an alternate to the class. For example, the MX Button class is marked with the `[Alternative]` metadata tag to indicate that you should use the Spark Button class instead.

Insert the `[Alternative]` metadata tag before the class definition of the class to be replaced. The `[Alternative]` metadata tag has the following syntax:

```
[Alternative(replacement="packageAndClassName", since="versionNum")]
```

The `replacement` option specifies the package and class name of the alternate class, and the `since` option specifies a version number.

ArrayElementType metadata tag

When you define an Array variable in ActionScript, you specify `Array` as the data type of the variable. However, you cannot specify the data type of the elements of the Array.

To allow the Flex MXML compiler to perform type checking on Array elements, you can use the `[ArrayElementType]` metadata tag to specify the allowed data type of the Array elements, as the following example shows:

```
public class MyTypedArrayComponent extends VBox {

    [ArrayElementType("String")]
    public var newStringProperty:Array;

    [ArrayElementType("Number")]
    public var newNumberProperty:Array;
    ...
}
```

***Note:** The MXML compiler checks for proper usage of the Array only in MXML code; it does not check Array usage in ActionScript code.*

In this example, you specify String as the allowed data type of the Array elements. If a user attempts to assign elements of a data type other than String to the Array in an MXML file, the compiler issues a syntax error, as the following example shows:

```
<MyComp:MyTypedArrayComponent>
    <MyComp:newStringProperty>
        <fx:Number>94062</fx:Number>
        <fx:Number>14850</fx:Number>
        <fx:Number>53402</fx:Number>
    </MyComp:newStringProperty>
</MyComp:MyTypedArrayComponent>
```

In this example, you try to use Number objects to initialize the Array, so the compiler issues an error.

You can also specify Array properties as tag attributes, rather than using child tags, as the following example shows:

```
<MyComp:MyTypedArrayComponent newNumberProperty="[abc,def]"/>
```

This MXML code generates an error because Flex cannot convert the Strings "`abc`" and "`def`" to a Number.

You insert the `[ArrayElementType]` metadata tag before the variable definition. The tag has the following syntax:

```
[ArrayElementType("elementType")]
```

The following table describes the property of the `[ArrayElementType]` metadata tag:

Property	Type	Description
`elementType`	String	Specifies the data type of the Array elements, and can be one of the ActionScript data types, such as String, Number, class, or interface. You must specify the type as a fully qualified class name, including the package.

Bindable metadata tag

When a property is the source of a data binding expression, Flex automatically copies the value of the source property to any destination property when the source property changes. To signal to Flex to perform the copy, you must use the `[Bindable]` metadata tag to register the property with Flex, and the source property must dispatch an event.

The `[Bindable]` metadata tag has the following syntax:

```
[Bindable]
[Bindable(event="eventname")]
```

If you omit the event name, Flex automatically creates an event named `propertyChange`.

For more information on data binding and on this metadata tag, see “Data binding” on page 299.

Working with bindable property chains

When you specify a property as the source of a data binding, Flex monitors not only that property for changes, but also the chain of properties leading up to it. The entire chain of properties, including the destination property, is called a *bindable property chain*. In the following example, `firstName.text` is a bindable property chain that includes both a `firstName` object and its `text` property:

```
<first>{firstName.text}</first>
```

You should raise an event when any named property in a bindable property chain changes. If the property is marked with the `[Bindable]` metadata tag, the Flex compiler generates the event for you.

The following example uses the `[Bindable]` metadata tag for a variable and a getter property. The example also shows how to call the `dispatchEvent()` function.

```
[Bindable]
public var minFontSize:Number = 5;

[Bindable("textChanged")]
public function get text():String {
     return myText;
}

public function set text(t : String):void {
     myText = t;
    dispatchEvent( new Event( "textChanged" ) );}
```

If you omit the event name in the `[Bindable]` metadata tag, the Flex compiler automatically generates and dispatches an event named `propertyChange` so that the property can be used as the source of a data binding expression.

You should also provide the compiler with specific information about an object by casting the object to a known type. In the following example, the `myList` List control contains Customer objects, so the `selectedItem` property is cast to a Customer object:

```
<fx:Model id="selectedCustomer">
    <customer>
        <name>{Customer(myList.selectedItem).name}</name>
        <address>{Customer(myList.selectedItem).address}</address>
        ...
    </customer>
</fx:Model>
```

There are some situations in which binding does not execute automatically as expected. Binding does not execute automatically when you change an entire item of a `dataProvider` property, as the following example shows:

```
dataProvider[i] = newItem
```

Binding also does not execute automatically for subproperties of properties that have `[Bindable]` metadata, as the following example shows:

```
...
[Bindable]
var temp;
// Binding is triggered:
temp = new Object();
// Binding is not triggered, because label not a bindable property
// of Object:
temp.label = foo;
...
```

In this code example, the problem with `{temp.label}` is that temp is an Object. You can solve this problem in one of the following ways:

- Preinitialize the Object.
- Assign an ObjectProxy to `temp`; all of an ObjectProxy's properties are bindable.
- Make `temp` a strongly typed object with a `label` property that is bindable.

Binding also does not execute automatically when you are binding data to a property that Flash Player updates automatically, such as the `mouseX` property.

The executeBindings() method of the UIComponent class executes all the bindings for which a UIComponent object is the destination. All containers and controls, as well as the Repeater component, extend the UIComponent class. The executeChildBindings() method of the Container and Repeater classes executes all of the bindings for which the child UIComponent components of a Container or Repeater class are destinations. All containers extend the Container class.

These methods give you a way to execute bindings that do not occur as expected. By adding one line of code, such as a call to executeChildBindings() method, you can update the user interface after making a change that does not cause bindings to execute. However, you should only use the executeBindings() method when you are sure that bindings do not execute automatically.

DefaultProperty metadata tag

The [DefaultProperty] metadata tag defines the name of the default property of the component when you use the component in an MXML file.

The [DefaultProperty] metadata tag has the following syntax:

```
[DefaultProperty("propertyName")]
```

The *propertyName* property specifies the name of the default property.

You can use the [DefaultProperty] metadata tag in your ActionScript component to define a single default property. For more information and an example, see “Creating a default property” on page 2439.

Deprecated metadata tag

A class or class element marked as deprecated is one which is considered obsolete, and whose use is discouraged in the current release. While the class or class element still works, its use can generate compiler warnings.

The mxmlc command-line compiler supports the show-deprecation-warnings compiler option, which, when true, configures the compiler to issue deprecation warnings when your application uses deprecated elements. The default value is true.

Insert the [Deprecated] metadata tag before a property, method, or class definition to mark that element as deprecated. The [Deprecated] metadata tag has the following options for its syntax when used with a class, property or method:

```
[Deprecated("string_describing_deprecation")]
[Deprecated(message="string_describing_deprecation")]
[Deprecated(replacement="string_specifying_replacement")]
[Deprecated(replacement="string_specifying_replacement", since="version_of_replacement")]
```

The following uses the [Deprecated] metadata tag to mark the dataProvider property as obsolete:

```
[Deprecated(replacement="MenuBarItem.data")]
public function set dataProvider(value:Object):void
{
    ...
}
```

The [Event], [Effect] and [Style] metadata tags also support deprecation. These tags support the following options for syntax:

```
[Event(... , deprecatedMessage="string_describing_deprecation")]
[Event(... , deprecatedReplacement="change2")]
[Event(... , deprecatedReplacement="string_specifying_replacement",
deprecatedSince="version_of_replacement")]
```

These metadata tags support the `deprecatedReplacement` and `deprecatedSince` attributes to mark the event, effect, or style as deprecated.

Effect metadata tag

The `[Effect]` metadata tag defines the name of the MXML property that you use to assign an effect to a component and the event that triggers the effect. If you define a custom effect, you can use the `[Effect]` metadata tag to specify that property to the Flex compiler.

For more information on defining custom effects, see "Custom effects" on page 2525.

An effect is paired with a trigger that invokes the effect. A *trigger* is an event, such as a mouse click on a component, a component getting focus, or a component becoming visible. An *effect* is a visible or audible change to the component that occurs over a period of time.

You insert the `[Effect]` metadata tag before the class definition in an ActionScript file or in the `<fx:Metadata>` block in an MXML file. The `[Effect]` metadata tag has the following syntax:

```
[Effect(name="eventNameEffect", event="eventName")]
```

The following table describes the properties of the `[Effect]` metadata tag:

Property	Type	Description
`eventNameEffect`	String	Specifies the name of the effect.
`eventName`	String	Specifies the name of the event that triggers the effect.

The `[Effect]` metadata tag is often paired with an `[Event]` metadata tag, where the `[Event]` metadata tag defines the event corresponding to the effect's trigger. By convention, the name of the effect is the event name with the suffix `Effect`, as the following example of an ActionScript file shows:

```
// Define event corresponding to the effect trigger.
[Event(name="darken", type="flash.events.Event")]
// Define the effect.
[Effect(name="darkenEffect", event="darken")]
class ModalText extends TextArea {
    ...
}
```

In an MXML file, you can define the event and effect in an `<fx:Metadata>` block, as the following example shows:

```
<fx:Metadata>
    [Event(name="darken", type="flash.events.Event")]
    [Effect(name="darkenEffect", event="darken")]
</fx:Metadata>
```

Event metadata tag

Use the `[Event]` metadata tag to define the MXML property for an event and the data type of the event object that a component emits. You insert the [Event] metadata tag before the class definition in an ActionScript file, or in the `<fx:Metadata>` block in an MXML file.

For more information on defining custom events, see "Custom events" on page 2369.

The `[Event]` metadata tag has the following syntax:

```
[Event(name="eventName", type="package.eventType")]
```

The following table describes the properties of the `[Event]` metadata tag:

Property	Type	Description
eventName	String	Specifies the name of the event.
eventType	String	Specifies the package and class that defines the data type of the event object. It is either the base event class, Event, or a subclass of the Event class. You must include the package in the class name.

The following example identifies the `myClickEvent` event as an event that the component can dispatch:

```
[Event(name="myClickEvent", type="flash.events.Event")]
```

If you do not identify an event in the class file with the `[Event]` metadata tag, the MXML compiler generates an error if you try to use the event name in MXML. Any component can register an event listener for the event in ActionScript by using the `addEventListener()` method, even if you omit the `[Event]` metadata tag.

The following example identifies the `myClickEvent` event as an event that an ActionScript component can dispatch:

```
[Event(name="myEnableEvent", type="flash.events.Event")]
public class MyComponent extends UIComponent
{
    ...
}
```

The following example shows the `[Event]` metadata tag in the `<fx:Metadata>` tag in an MXML file:

```
<?xml version="1.0"?>
<!-- TextAreaEnabled.mxml -->
<mx:TextArea xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Metadata>
        [Event(name="myEnableEvent", type="flash.events.Event")]
    </fx:Metadata>

    ....

</mx:TextArea>
```

HostComponent metadata tag

Use the `[HostComponent]` metadata tag to identify the host component of a Spark skin class. The `[HostComponent]` metadata tag has the following syntax:

```
<Metadata>
    [HostComponent(componentName)]
</Metadata>
```

For example:

```
<Metadata>
    [HostComponent("spark.components.Button")]
</Metadata>
```

As a result of this metadata, Flex creates the property `hostComponent` on the skin class. You can then use this property to access public members of the host component's instance from within the skin. For example, in a Button skin, you can access the Button's style properties.

For more information, see "Spark Skinning" on page 1602.

IconFile metadata tag

Use the [IconFile] metadata tag to identify the filename for the icon that represents the component in the Insert bar of Flash Builder.

The [IconFile] metadata tag has the following syntax:

```
[IconFile("fileName")]
```

The fileName property specifies a PNG, GIF, or JPEG file that contains the icon, as the following example shows:

```
[IconFile("MyButton.png")]
public class MyButton extends Button
{
    ...
}
```

Inspectable metadata tag

The [Inspectable] metadata tag defines information about an attribute of your component that you expose in code hints and in the Property inspector area of Flash Builder. The [Inspectable] metadata tag is not required for either code hints or the Property inspector. The following rules determine how Flash Builder displays this information:

- All public properties in components appear in code hints and in the Flash Builder Property inspector. If you have extra information about the property that you want to add, such as enumeration values or that a String property represents a file path, add the [Inspectable] metadata tag with that information.
- Code hints for components and the information in the Property inspector come from the same data. Therefore, if the attribute appears in one, it should appear in the other.
- Code hints for ActionScript components do not require metadata to work correctly so that you always see the appropriate code hints, depending the current scope. Flash Builder uses the public, protected, private, and static keywords, plus the current scope, to determine which ActionScript code hints to show.

The [Inspectable] metadata tag must immediately precede the property's variable declaration or the setter and getter methods to be bound to that property. The [Inspectable] metadata tag has the following syntaxes:

```
[Inspectable(attribute=value[,attribute=value,...])]
property_declaration name:type;
[Inspectable(attribute=value[,attribute=value,...])]
setter_getter_declarations;
```

The following table describes the properties of the [Inspectable] metadata tag:

Property	Type	Description
category	String	Groups the property into a specific subcategory in the Property inspector of the Flash Builder user interface. The default category is "Other". Specify a value of "Common", "Effects", "Events", "Layout Constraints", "Size", "Styles", "Text", or "Other".
defaultValue	String or Number	Sets the initial value in the editor that appears in the Property inspector when you modify the attribute. The default value is determined from the property definition.
enumeration	String	Specifies a comma-delimited list of legal values for the property. Only these values are allowed; for example, item1,item2,item3. Notice the lack of a space character between items so that Flex Builder does not interpret a space as a part of a valid value. This information appears as code hints and in the Property inspector. If you define a Boolean variable, Flash Builder automatically shows true and false without you having to specifying them using enumeration.

Property	Type	Description
environment	String	Specifies which inspectable properties should not be allowed (environment=none), which are used only for Flash Builder (environment=Flash), and which are used only by Flex and not Flash Builder (environment=MXML).
format	String	Determines the type of editor that appears in the Property inspector when you modify the attribute. You can use this property when the data type of the attribute is not specific to its function. For example, for a property of type Number, you can specify format="Color" to cause Flash Builder to open a color editor when you modify the attribute. Common values for the format property include "Length", "Color", "Time", "EmbeddedFile", and "File".
listOffset	Number	Specifies the default index into a List value.
name	String	Specifies the display name for the property; for example, FontWidth. If not specified, use the property's name, such as _fontWidth.
type	String	Specifies the type specifier. If omitted, use the property's type. The following values are valid: • Array • Boolean • Color • Font Name • List • Number • Object • String If the property is an Array, you must list the valid values for the Array.
variable	String	Specifies the variable to which this parameter is bound.
verbose	Number	Indicates that this inspectable property should be displayed in the Flash Builder user interface only when the user indicates that verbose properties should be included. If this property is not specified, Flash Builder assumes that the property should be displayed.

The following example defines the myProp parameter as inspectable:

```
[Inspectable(defaultValue=true, verbose=1, category="Other")]
public var myProp:Boolean;
```

InstanceType metadata tag

The [InstanceType] metadata tag specifies the allowed data type of a property of type IDeferredInstance, as the following example shows:

```
// Define a deferred property for the top component.
[InstanceType("mx.controls.Label")]
public var topRow:IDeferredInstance;
```

The compiler validates that users assign values only of the specified type to the property. In this example, if the component user sets the topRow property to a value of a type other than mx.controls.Label, the compiler issues an error message.

You use the `[InstanceType]` metadata tag when creating template components. For more information, see "Template components" on page 2507.

The `[InstanceType]` metadata tag has the following syntax:

```
[InstanceType("package.className")]
```

You must specify a fully qualified package and class name.

NonCommittingChangeEvent metadata tag

The `[NonCommittingChangeEvent]` metadata tag identifies an event as an interim trigger, which means that the event should not invoke Flex data validators on the property. You use this tag for properties that might change often, but which you do not want to validate on every change.

An example of this is if you tied a validator to the `text` property of a TextInput control. The `text` property changes on every keystroke, but you do not want to validate the property until the user presses the Enter key or changes focus away from the field. The `NonCommittingChangeEvent` tag lets you dispatch a change event, but that does not trigger validation.

You insert the `[NonCommittingChangeEvent]` metadata tag before an ActionScript property definition or before a setter or getter method. The `[NonCommittingChangeEvent]` metadata tag has the following syntax:

```
[NonCommittingChangeEvent("event_name")]
```

In the following example, the component dispatches the `change` event every time the user enters a keystroke, but the `change` event does not trigger data binding or data validators. When the user completes data entry by pressing the Enter key, the component broadcasts the `valueCommit` event to trigger any data bindings and data validators:

```
[Event(name="change", type="flash.events.Event")]
class MyText extends UIComponent {
    ...

    [Bindable(event="valueCommit")]
    [NonCommittingChangeEvent("change")]
    function get text():String {
        return getText();
    }
    function set text(t):void {
        setText(t);
        // Dispatch events.
    }
}
```

RemoteClass metadata tag

Use the `[RemoteClass]` metadata tag to register the class with Flex so that Flex preserves type information when a class instance is serialized by using Action Message Format (AMF). You insert the `[RemoteClass]` metadata tag before an ActionScript class definition.

The `[RemoteClass]` metadata tag has the following syntax:

```
[RemoteClass]
```

You can also use this tag to represent a server-side Java object in a client application. You use the `[RemoteClass(alias=" ")]` metadata tag to create an ActionScript object that maps directly to the Java object. You specify the fully qualified class name of the Java class as the value of `alias`. For more information, see Accessing Server-Side Data with Flex.

RichTextContent metadata tag

If a property is typed as a String, the compiler automatically tries to convert its value in MXML to a String. If the property is of a type such as *, Object, or Array, the compiler by default attempts to convert the value to an appropriate data type. Use the `[RichTextContent]` metadata tag to indicate that the value of a property in MXML should always be interpreted by the compiler as a String.

For example, the `content` property of the spark.components.TextArea and spark.primitives.RichText classes is typed as Object. However, these classes use the `[RichTextContent]` metadata tag to indicate that the value of the property in MXML should always be interpreted as a String.

Shown below are two examples that set the `content` property of the RichText class:

```
<!-- Without [RichTextContent] metadata, interpret the value as type Number. -->
<s:RichText>
    <s:content>1</s:content>
</s:RichText>

<!-- Without [RichTextContent] metadata, interpret the value as type Array. -->
<s:RichText>
    <s:content>[1]</s:content>
</s:RichText>
```

Data binding syntax, { }, and @ functions, such as `@Resource` and `@Embed`, are supported by properties that use the `[RichTextContent]` metadata tag.

You insert the `[RichTextContent]` metadata tag a property before a property definition in ActionScript. The `[RichTextContent]` metadata tag has the following syntax:

```
[RichTextContent]
```

SkinPart metadata tag

Components can uses skins made up of skin parts. Use the `[SkinPart]` metadata tag to define a property of a component that corresponds to a skin part. Users of the component do not set the skin part properties directly. The component's skin sets the skin part properties.

Insert the `[SkinPart]` metadata tag before any property that corresponds to a skin part. The `[SkinPart]` metadata tag has the following syntax:

```
[SkinPart(type="package.className", required="true_false")]
/**
* Optional ASDoc comment.
*/
Property definition
```

The `type` and `required` attributes are optional. The `type` attribute specifies the data type of the skin part, which determines whether the part is static or dynamic. The default value of `type` is determined by the data type of the property.

The `required` attribute specifies if the skin class must define the skin part. The default value of the `required` attribute is `false`.

SkinPart metadata is inherited by subclasses of the component.

For more information, see “Spark Skinning” on page 1602.

Static skin parts

Static skin parts are created once by an instance of a component, and are defined as shown below:

```
[SkinPart]
/**
* ASDoc comment for thumb.
*/
public var thumb:spark.components.Button;
```

The data type for static parts is the data type of the part property. In this example above, the type is Button. Therefore, static skin parts typically omit the `type` attribute of the `[SkinPart]` metadata tag.

Dynamic skin parts

Some components create multiple instances of a skin part. For example, the Spark ButtonBar control can create any number of buttons. Dynamic skin parts can be created multiple times by a component. The data type of a dynamic skin part property is always IFactory, but the metadata tag can optionally define the data type of the skin part by using the `type` property.

For example from the spark.components.ButtonBar class:

```
[SkinPart(required="false", type="mx.core.IVisualElement")]
/**
* A skin part that defines the first button.
*/
public var firstButton:IFactory;
```

Because the data type of the skin part is IFactory, it is a dynamic skin part. Each instance of the skin part is of type mx.core.IVisualElement.

SkinState metadata tag

The `[SkinState]` metadata tag defines the view states that a component's skin must support. The tag goes outside the component's class definition, and inside the package definition. The tag is inherited, but can be overridden in a subclass.

The SkinState tag has the following ActionScript syntax:

```
[SkinState("stateName")]
```

The following example defines two skin states for a component:

```
package spark.components.supportCl asses
{
    /**
    *  Optional ASDoc comment. */
    [SkinState("n ormal")]

    /**
    *  Optional ASDoc comment. */
    [SkinState("disabled")]

    public class MyClass {}
```

For more information, see "Spark Skinning" on page 1602.

Style metadata tag

Use the `[Style]` metadata tag to define the MXML tag attribute for a style property for the component. You insert the `[Style]` metadata tag before the class definition in an ActionScript file, or in the `<fx:Metadata>` block in an MXML file.

The `[Style]` metadata tag has the following syntax:

```
[Style(name="style_name"[,property="value",...])]
```

The following table describes the properties for the `[Style]` metadata tag:

Option	Type	Description
`name`	String	(Required) Specifies the name of the style.
`arrayType`	String	If `type` is `Array`, `arrayType` specifies the data type of the Array elements. If the data type is not an ActionScript type such as Number or Date, use a qualified class name in the form *packageName.className*.
`enumeration`	String	Specifies an enumerated list of possible values for the style property.
`format`	String	Specifies the units of the property. For example, if you specify `type` as `"Number"` , you might specify `format="Length"` if the style defines a length measured in pixels. Or, if you specify `type="uint"`, you might set `format="Color"` if the style defines an RGB color. The possible values are `Boolean`, `Color`, `Number`, `Length`, `uint`, `Time`, `File`, `EmbeddedFile`, `int`, `ICollectionView`, `Array`, `Class`, `String`, and `Object`.
`inherit`	String	Specifies whether the property is inheriting. Valid values are `yes` and `no`. This property refers to CSS inheritance, not object-oriented inheritance. All subclasses automatically use object-oriented inheritance to inherit the style property definitions of their superclasses. Some style properties are inherited using CSS inheritance. If you set an inheritable style property on a parent container, its children inherit that style property. For example, if you define `fontFamily` as Times for a Panel container, all children of that container will also use Times for `fontFamily`, unless they override that property. If you set a noninheritable style, such as `textDecoration`, on a parent container, only the parent container and not its children use that style. For more information on inheritable style properties, see “About style inheritance” on page 1526.
`states`	String	For skin properties, specifies that you can use the style to specify a stateful skin for multiple states of the component. For example, the definition of the `Slider.thumbSkin` style uses the following `[Style]` metadata tag: `[Style(name="thumbSkin", type="Class", inherit="no", states="disabled, down, over, up")]` This line specifies that you can use the `Slider.thumbSkin` style to specify a stateful skin for the disabled, down, over, and up states of the Slider control. For more information, see “Skinning MX components” on page 1655.
`theme`	String	If the style is only valid for a specific theme, specifies the name of the theme. For example, some styles on Flex components are only valid if you are using the Spark or Halo theme. For more information, see “About themes” on page 1561
`type`	String	Specifies the data type of the value that you write to the style property. If the type is not an ActionScript type such as Number or Date, use a qualified class name in the form *packageName.className*.

The following example shows the definition of the `textSelectedColor` style property:

```
[Style(name="textSelectedColor",type="Number",format="Color",inherit="yes")]
```

The next example shows the definition of the `verticalAlign` style property:

```
[Style(name="verticalAlign", type="String", enumeration="bottom,middle,top", inherit="no")]
```

For more information on the `[Style]` metadata tag, see “Custom style properties” on page 2501.

SWF metadata tag

Use the `[SWF]` metadata tag to specify attributes of the application when you write the main application file in ActionScript. Typically, you set these attributes by using the `<s:Application>` tag when you write the main application file in MXML.

The `[SWF]` tag has the following syntax:

```
[SWF(option="value"[,option="value",...])]
```

You can specify several options the `[SWF]` metadata tag. The following table describes these options:

Option	Type	Description
`frameRate`	Number	Specifies the frame rate of the application, in frames per second. The default value is 24.
`height`	Number	The height of the application, in pixels.
`heightPercent`	Number	The height of the application, as a percentage of the browser window. Include the percent sign (`%`) in the value.
`pageTitle`	String	Specifies a String that appears in the title bar of the browser. This property provides the same functionality as the HTML `<title>` tag.
`scriptRecursionLimit`	Number	Specifies the maximum depth of Flash Player or AIR call stack before Flash Player or AIR stops. This is essentially the stack overflow limit. The default value is 1000.
`scriptTimeLimit`	Number	Specifies the maximum duration, in seconds, that an ActionScript event listener can execute before Flash Player or AIR assumes that it has stopped processing and aborts it. The default value is 60 seconds, which is also the maximum allowable value that you can set.
`width`	Number	The width of the application, in pixels.
`widthPercent`	Number	The width of the application, as a percentage of the browser window. Include the percent sign (`%`) in the value.

For more information on these options, see "Specifying options of the Application container" on page 402.

Transient metadata tag

Use the `[Transient]` metadata tag to identifies a property that should be omitted from data that is sent to the server when an ActionScript object is mapped to a Java object using the `[RemoteClass]` metadata tag.

The `[Transient]` metadata tag has the following syntax:

```
[Transient]
public var count:Number = 5;
```

Component compilation

When you compile an application, you create a SWF file that a user can download and play. You can also compile any custom components that you create as part of the application.

When you create a component, you save it to a location that the Adobe® Flex® compiler can access. You can save your components as MXML and ActionScript files, as SWC files, or as Runtime Shared Libraries (RSLs). You have several options when you compile the components.

About compiling

You compile a custom component so that you can use it as part of your application. You can compile the component when you compile the entire application, or you can compile it separately so that you can link it into the application at a later time.

Flex component file types

When you create a Flex component, you can distribute it in one of several different file formats, as the following table shows:

File format	Extension	Description
MXML	.mxml	A component implemented as an MXML file.
ActionScript	.as	A component implemented as an ActionScript class.
SWC	.swc	A component implemented as an MXML or ActionScript file, and then packaged as a SWC file. A SWC file contains components that you package and reuse among multiple applications. The SWC file is then compiled into your application when you create the application's SWF file.
RSL	.swc	A component implemented as an MXML or ActionScript file, and then deployed as an RSL. An RSL is a stand-alone file that is downloaded separately from your application's SWF file, and cached on the client computer for use with multiple application SWF files.

You must take into consideration the file format and file location when you compile an application that uses the component.

About compiling with Flex SDK

Adobe Flex includes two compilers: mxmlc and compc. You use the mxmlc compiler to compile MXML, ActionScript, SWC, and RSL files into a single SWF file. After your application is compiled and deployed on your web or application server, a user can make an HTTP request to download and play the SWF file on their computer.

You use the compc compiler to compile components, classes, and other files into SWC files or into RSLs.

You can use the compc and mxmlc compilers from Adobe Flash Builder or from a command line. For more information on using the compilers, see "Flex compilers" on page 2164.

The most basic example of using the mxmlc compiler is one in which the MXML file has no external dependencies (such as components in a SWC file or ActionScript classes). In this case, you open mxmlc and point it to your MXML file:

```
$ mxmlc c:/myfiles/app.mxml
```

The default option is the target file to compile into a SWF file, and it must have a value. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file, as in the following example:

```
$ mxmlc -option arg1 arg2 arg3 -- target_file.mxml
```

About case sensitivity during a compilation

The Flex compilers use a case-sensitive file lookup on all file systems. On case-insensitive file systems, such as the Macintosh and Windows file systems, the Flex compiler generates a case-mismatch error when you use a component with the incorrect case. On case-sensitive file systems, such as the UNIX file system, the Flex compiler generates a component-not-found error when you use a component with the incorrect case.

About the ActionScript source path

Typically, you put component files in directories that are in the ActionScript source path. These include your application's root directory, its subdirectories, and any directory that you specify to the compiler. To specify a directory, you use the `source-path` option to the mxmlc compiler, or the Project Properties dialog box in Flash Builder.

The following rules can help you organize your custom components:

- An application can access MXML and ActionScript components in the same directory and in its subdirectories.
- An ActionScript component in a subdirectory of the main application directory must define a fully qualified package name that is relative to the location of the application's root directory. For example, if you define a custom component in the dir1/dir2/myControls/PieChart.as file, its fully qualified package name must be dir1.dir2.myControls, assuming dir1 is an immediate subdirectory of the main application directory.
- An MXML component does not include a package name definition. However, you must declare a namespace definition in the file that references the MXML component that corresponds to the directory location of the MXML component, either in a subdirectory of the application's root directory or in a subdirectory of the source path. For more information, see "Simple MXML components" on page 2399.
- An application can access MXML and ActionScript components in the directories included in the ActionScript source path. The component search order in the source path is based on the order of the directories listed in the source path.
- An ActionScript component in a subdirectory of a directory included in the source path must define a fully qualified package name that is relative to the location of the source path directory. For example, if you define a custom component in the file dir1/dir2/myControls/PieChart.as, and dir1 is included in the ActionScript source path, its fully qualified package name must be dir1.dir2.myControls.
- The `<fx:Script>` tag in the main MXML file, and in dependent MXML component files, can reference components located in the ActionScript source path.

Compiling components with Flex SDK

How you compile an application with Adobe Flex SDK is based on how you distribute your custom components that are distributed as MXML, ActionScript, SWC, and RSL files.

Distributing components as MXML and ActionScript files

When you compile an application with Flex SDK, you define where the MXML and ActionScript files for your custom components exist in the directory structure of your application, or in the directory structure of components shared by multiple applications.

For example, you can create a component for use by a single application. In that case, you store it in the directory structure of the application, usually in a subdirectory under the directory that contains the main file of the application. The component is then compiled with the entire application into the resultant SWF file.

You can also create a component that is shared among multiple applications as an MXML or ActionScript file. In that case, store the component in a location that is included in the ActionScript source path of the application. When Flex compiles the application, it also compiles the components included in the application's ActionScript source path.

You specify the directory location of the shared components by using one of the following methods:

Flash Builder Open the Project Properties dialog box, and then select Flex Build Path to set the ActionScript source path.

mxmlc compiler Use the `source-path` option to the mxmlc compiler to specify the directory location of your shared MXML and ActionScript files.

Distributing components as SWC files

A SWC file is an archive file of Flex components. SWC files make it easy to exchange components among Flex developers. You need to exchange only a single file, rather than the MXML or ActionScript files, images, and other resource files. In addition, the SWF file inside a SWC file is compiled, which means that the code is hidden from casual view. Finally, compiling a component as a SWC file can make namespace allocation an easier process.

SWC files can contain one or more components and are packaged and expanded with the PKZIP archive format. You can open and examine a SWC file by using WinZip, JAR, or another archiving tool. However, do not manually change the contents of a SWC file, and do not try to run the SWF file that is in a SWC file outside of the SWC file.

When you compile your application, you specify the directory location of the SWC files by using one of the following methods:

Flash Builder Open the Project Properties dialog box, and then select Flex Build Path to set the library directories that contain the SWC files.

mxmlc compiler Set the `library-classpath` option to the mxmlc compiler to specify the directory location of your SWC files.

One of the advantages of distributing components as SWC files is that you can define a global style sheet, named defaults.css, in the SWC file. The defaults.css file defines the default style settings for all of the components defined in the SWC file. For more information, see "Applying styles from a defaults.css file" on page 2409.

For more information about SWC files, see "Flex compilers" on page 2164, and Basic workflow to develop an application with Flex.

Distributing components as RSLs

One way to reduce the size of your application's SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but must be transferred to the client only once. These shared files are known as Runtime Shared Libraries (RSLs).

If you have multiple applications but those applications share a core set of components or classes, your users download those assets only once as an RSL. The applications that share the assets in the RSL use the same cached RSL as the source for the libraries as long as they are in the same domain. By using an RSL, you can reduce the resulting file size for your applications. The benefits increase as the number of applications that use the RSL increases. If you only have one application, putting components into RSLs does not reduce the aggregate download size, and may increase it.

When you compile your application, you specify the directory location of an RSL file by using one of the following methods:

Flash Builder Open the Project Properties dialog box, and then select Flex Build Path to set the library directories that contain the SWC files.

mxmlc compiler Set the `external-library-path` option to the mxmlc compiler to specify the location of the RSL file at compile time. Set the `runtime-shared-libraries` option to the mxmlc compiler to specify the relative location of the RSL file when the application is deployed.

For more information, including information on how to create an RSL file, see "Runtime Shared Libraries" on page 253.

Example: Compiling a custom formatter component

The example in this section uses a custom formatter component that is defined as an ActionScript file. The name of the formatter is MySimpleFormatter, and it is defined in the file MySimpleFormatter.as. For more information on creating customer formatter components, see "Custom formatters" on page 2513.

The process for compiling an MXML file is the same as for an ActionScript file. For an example of deploying an MXML file, see "Simple MXML components" on page 2399.

Distributing a component as an ActionScript file

When you distribute a component defined as an ActionScript file, you can store it within the same directory structure as your application files or in a directory specified in the ActionScript source path.

The MXML tag name for a custom component consists of two parts: the namespace prefix and the tag name. The namespace prefix tells Flex where to look for the file that implements the custom component. The tag name corresponds to the filename of the component, in this case MySimpleFormatter.as. Therefore, the file MySimpleFormatter.as defines a component with the tag name of `<`*namespace*`:MySimpleFormatter>`.

The main application MXML file defines the namespace prefix used to reference the component in the `<s:Application>` tag. When you deploy your formatter as an ActionScript file, you refer to it in one of the following ways:

1 If you store the formatter component in the same directory as the application file, or in a directory that the ActionScript source path (not a subdirectory) specifies, you define the formatter by using an empty `package` statement, as the following example shows:

```
package
{
    //Import base Formatter class.
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

You can refer to it as the following example shows. In the following code, the local namespace (*) is mapped to the prefix MyComp.

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">

    <MyComp:MySimpleFormatter/>

</s:Application>
```

If the same file exists in the ActionScript source path directory and the application directory, Flex uses the file in the application directory.

2 If you store the formatter component in a subdirectory of the directory that contains the application file, you specify that directory as part of the package statement, as the following example shows:

```
package myComponents.formatters
{
    //Import base Formatter class
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

In this example, the MySimpleFormatter.as file is located in the myComponents/formatter subdirectory of the main application directory. You map the myComponents.formatters namespace to the MyComp prefix, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.formatters.*">

    <MyComp:MySimpleFormatter/>

</s:Application>
```

If multiple files with the same name exist under an ActionScript source path subdirectory and the application subdirectory, Flex uses the file under the application subdirectory.

3 If you store the formatter component in a subdirectory of the ActionScript source path directory, you specify that subdirectory as part of the package statement, as the following example shows:

```
package flexSharedRoot.custom.components
{
    //Import base Formatter class.
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

You then use a namespace that specifies the subdirectory. The following code declares a component that is in the flexSharedRoot/custom/components directory:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="flexSharedRoot.custom.components.*"/>

    <MyComp:MySimpleFormatter/>

</s:Application>
```

If the same file exists in the ActionScript source path directory and the application directory, Flex uses the file in the application file directory.

Distributing a component as a SWC file

To create a SWC file, use the compc compiler in the *flex_install_dir*/bin directory. The compc compiler generates a SWC file from MXML component source files and/or ActionScript component source files.

In this example, you create a SWC file for a custom formatter component that you defined by using the following package and class definition:

```
package myComponents.formatters
{
    //Import base Formatter class.
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

In this example, the MySimpleFormatter.as file is in the directory c:\flex\myComponentsForSWCs\myComponents\formatters.

You use the following compc command from the *flex_install_dir*/bin directory to create the SWC file for this component:

```
.\compc -source-path c:\flex\myComponentsForSWCs\
    -include-classes myComponents.formatters.MySimpleFormatter
    -o c:\flex\mainApp\MyFormatterSWC.swc
```

In this example, you use the following options of the compc compiler:

Option name	Description
`-source-path`	Specifies the base directory location of the MySimpleFormatter.as file. It does not include the directories that the component's package statement defines.
`-include-classes`	Specifies classes to include in the SWC file. You provide the class name (MyComponents.formatters.MySimpleFormatter) not the filename of the source code. All classes specified with this option must be in the compiler's source path, which is specified in the `source-path` compiler option. You can use packaged and unpackaged classes. To use components in namespaces, use the `include-namespaces` option. If the components are in packages, use dot (.) notation rather than slashes to separate package levels.
`-o`	Specifies the name and directory location of the output SWC file. In this example, the directory is c:\flex\mainApp, the directory that contains your main application.

In your main application file, you specify the component's namespace, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.formatters.*">

    <MyComp:MyFormatter/>

</s:Application>
```

When you distribute SWC files, ensure that the corresponding ActionScript file is not in the directory structure of the application or in the ActionScript source path. Otherwise, Flex might use the ActionScript file, rather than the SWC file.

When you use mxmlc to compile the main application, ensure that the c:\flex\mainApp directory is included in the library path; otherwise, mxmlc cannot locate the SWC file.

For more information about SWC files, see "Flex compilers" on page 2164, and Build projects.

Distributing a component as an RSL file

You create an RSL by using the compc tool, and then pass the library's location to the compiler when you compile your application. For more information, including an example, see "Runtime Shared Libraries" on page 253.

Simple MXML components

Applications for Adobe® Flex® typically consist of multiple MXML and ActionScript files, and each MXML file is a separate MXML component. MXML components let you encapsulate functionality in a reusable component, extend an existing Flex component by adding new functionality to it, and reference the MXML component by using an MXML tag.

For information on advanced techniques for creating MXML components, see "Advanced MXML components" on page 2412.

About MXML components

In typical applications, you do not code the entire application within a single source code file. Such an implementation makes it difficult for multiple developers to work on the project simultaneously, makes it difficult to debug, and discourages code reuse.

Instead, you develop applications by using multiple MXML and ActionScript files. This architecture promotes a modular design, code reuse, and lets multiple developers contribute to the implementation.

MXML components are MXML files that you reference by using MXML tags from within other MXML files. One of the main uses of MXML components is to extend the functionality of an existing Flex component.

For example, Flex supplies a ComboBox control that you can use as part of a form that collects address information from a customer. You can use a ComboBox to let the user select the State portion of the address from a list of the 50 states in the U.S. In an application that has multiple locations where a user can enter an address, it would be tedious to create and initialize multiple ComboBox controls with the information about all 50 states.

Instead, you create an MXML component that contains a ComboBox control with all 50 states defined within it. Then, wherever you must add a state selector to your application, you use your custom MXML component.

Creating MXML components

An application that uses MXML components includes a main MXML application file, which contains the `<s:Application>` root tag, and references one or more components that are defined in separate MXML and ActionScript files. Each MXML component extends an existing Flex component, or another MXML component.

You create an MXML component in an MXML file where the component's filename becomes its MXML tag name. For example, a file named StateComboBox.mxml defines a component with the tag name of `<StateComboBox>`.

The root tag of an MXML component is a component tag, either a Flex component or another MXML component. The root tag specifies the namespace definitions for the component. For example, the following MXML component extends the standard Flex ComboBox control.

```
<?xml version="1.0"?>
<!-- createcomps_mxml/StateComboBox.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
            <!-- Add all other states. -->
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

As part of its implementation, a custom MXML component can reference another custom MXML component.

The main application, or any other MXML component file, references the StateComboBox component, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MXMLMyApplication.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">
    <MyComp:StateComboBox/>
</s:Application>
```

In this example, the main application file includes a new namespace definition of `xmlns:MyComp="*"` as part of the `<s:Application>` tag. This namespace definition specifies the location of the MXML component. In this case, it specifies that the component is in the same directory as the main application file, or if you are using Adobe® LiveCycle™ Data Services ES, in the WEB-INF/flex/user-classes directory.

As a best practice, store your components in a subdirectory. For example, you can write this file to the myComponents directory, a subdirectory of your main application directory. For more information on the namespace, see “Developing applications in MXML” on page 4.

The StateComboBox.mxml file specifies the ComboBox control as its root tag, so you can reference all of the properties of the ComboBox control within the MXML tag of your custom component, or in the ActionScript specified within an `<fx:Script>` tag. For example, the following code specifies the `maxChars` property and a listener for the `close` event for your custom control:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MyApplicationProps.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:local="*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            private function handleCloseEvent(eventObj:Event):void {
                myTA.text="foo";
            }
        ]]>
    </fx:Script>
    <local:StateComboBox maxChars="25"
        close="handleCloseEvent(event);"/>
    <mx:TextArea id="myTA" />
</s:Application>
```

Limitation on the default property of the root tag

Many Flex components define a single default property. The *default property* is the MXML tag property that is implicit for content inside of the MXML tag if you do not explicitly specify a property. For example, consider the following MXML tag definition:

```
<s:SomeTag>
    anything here
</s:SomeTag>
```

If this tag defines a default property named `default_property`, the preceding tag definition is equivalent to the following code:

```
<s:SomeTag>
    <s:default_property>
        anything here
    </s:default_property>
</s:SomeTag>
```

However, the default property mechanism does not work for root tags of MXML components. In this situation, you must use child tags to define the default property, as the following example shows:

```
<?xml version="1.0"?>
<s:SomeTag xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo">
    <s:default_property>
        anything here
    </s:default_property>
</s:SomeTag>
```

MXML components and ActionScript classes

When you create a custom MXML component, you define a new ActionScript class where the class name corresponds to the filename of the MXML component. Your new class is a subclass of the component's root tag, and therefore inherits all of the properties and methods of the root tag. However, because you are defining the component in MXML, many of the intricacies of creating an ActionScript class are hidden from you.

For example, in "Creating MXML components" on page 2399, you defined the component StateComboBox.mxml by using the `<s:ComboBox>` tag as its root tag. Therefore, StateComboBox.mxml defines a subclass of the ComboBox class.

Creating composite MXML components

A composite MXML component is a component that contains multiple component definitions within it. To create a composite component, you specify a container as its root tag, and then add components as children of the container.

For example, the following component contains an address form created by specifying a Form container as the root tag of the component, and then defining several children of the Form container. One of the `<mx:FormItem>` tags contains a reference to the `<MyComp:StateComboBox>` tag that you created in "Creating MXML components" on page 2399:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/AddressForm.mxml -->
<s:Form xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">
    <s:FormItem label="NameField">
        <s:TextInput/>
    </s:FormItem>
    <s:FormItem label="Street">
        <s:TextInput/>
    </s:FormItem>
    <s:FormItem label="City" >
        <s:TextInput/>
    </s:FormItem>
    <s:FormItem label="State" >
        <MyComp:StateComboBox/>
    </s:FormItem>
</s:Form>
```

The following application file references the AddressForm component in the `<AddressForm>` tag:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MyApplicationAddressForm.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*" >
    <MyComp:AddressForm/>
</s:Application>
```

If you include child tags of the root container tag in an MXML component file, you cannot add child tags when you use the component as a custom tag in another MXML file. If you define an empty container in an MXML file, you can add child tags when you use the component as a custom tag.

***Note:** The restriction on child tags refers to the child tags that correspond to visual components. Visual components are subclasses of the UIComponent component. You can always insert tags for nonvisual components, such as ActionScript blocks, styles, effects, formatters, validators, and other types of nonvisual components, regardless of how you define your custom component.*

The following example defines an empty Form container in an MXML component:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/EmptyForm.mxml -->
<s:Form xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
/>
```

This component defines no children of the Form container; therefore, you can add children when you use it in another MXML file, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/MainEmptyForm.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">
    <MyComp:EmptyForm>
        <s:FormItem label="Name">
            <s:TextInput/>
        </s:FormItem>
    </MyComp:EmptyForm>
</s:Application>
```

The AddressForm.mxml file specifies the Form container as its root tag. Because you define a container as the root tag of the MXML component, you are creating a subclass of that container, and you can reference all of the properties and methods of the root tag when using your MXML component. Therefore, in the main application, you can reference all of the properties of the Form container in the MXML tag that corresponds to your custom component, or in any ActionScript code in the main application. However, you cannot reference properties of the children of the Form container.

For example, the following example sets the `fontSize` property and a listener for the `creationComplete` event for your custom control, but you cannot specify properties for the child CheckBox or TextInput controls of the Form container:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MainEmptyFormProps.mxml-->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">
    <fx:Script>
        <![CDATA[
            import mx.events.FlexEvent;
            private function handleCreationCompleteEvent(event:FlexEvent):void {
                // Handle creationComplete event.
            }
        ]]>
    </fx:Script>
    <MyComp:AddressForm fontSize="15"
        creationComplete="handleCreationCompleteEvent(event);"/>
</s:Application>
```

To configure the children of a custom MXML component, you define new properties in the MXML component, and then use those new properties to pass configuration information to the component children. For more information, see “Advanced MXML components” on page 2412.

Scoping in custom components

Scoping is mostly a description of what the `this` keyword refers to at any given point in your application. In an `<fx:Script>` tag in an MXML file, the `this` keyword always refers to the current scope. In the main application file, the file that contains the `<s:Application>` tag, the current scope is the Application object; therefore, the `this` keyword refers to the Application object.

In an MXML component, Flex executes in the context of the custom component. The current scope is defined by the root tag of the file. So, the `this` keyword refers not to the Application object, but to the object defined by the root tag of the MXML file.

For more information on scoping, see “Using ActionScript” on page 32.

The root tag of an MXML component cannot contain an `id` property. Therefore, if you refer to the object defined by the root tag in the body of the component, you must use the `this` keyword, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/myComponents/StateComboBoxThis.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    close="handleCloseEvent(event);">
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            // Define a property to hold the current index.
            public var stateIndex:Number;
            private function handleCloseEvent(eventObj:Event):void {
                stateIndex = this.selectedIndex;
            }
        ]]>
    </fx:Script>
    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

This example defines an event listener for the ComboBox control that updates the `stateIndex` property when the ComboBox control closes.

Applying styles to your custom component

Along with skins, styles define the look and feel of your Flex applications. You can use styles to change the appearance of a single component, or apply them across all components.

When working with custom components, you have several options for how you use styles. You can define your custom components so that they contain no style information at all. That design allows the application developer who is using your component to apply styles to match the rest of their application. For example, if you define a custom component to display text, the application developer can style it to ensure that the font, font size, and font style of your component match the rest of the application.

Alternatively, you might develop a component that you want to deploy with a built-in look so that it is not necessary for application developers to apply any additional styles to it. This type of component might be useful for applications that require a header or footer with a fixed look, while the body of the application has more flexibility in its look.

Or you might develop a custom component by using a combination of these approaches. This type of design lets application developers set some styles, but not others.

For general information on Flex styles, see "Styles and themes" on page 1492. For information on creating custom styles, see "Custom style properties" on page 2501.

Applying styles from the custom component

You can choose to define styles within your MXML component so that the component has the same appearance whenever it is used, and application developers do not have to worry about applying styles to it.

In the definition of your custom component, you define styles by using one or both of the following mechanisms:

- Tag properties
- Class selectors

The following custom component defines a style by using a tag property of the ComboBox control:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/myComponents/StateComboBoxWithStyleProps.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"

    fontWeight="bold"
    fontSize="15">
    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

Alternatively, you can define these styles by using a class selector style declaration, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/myComponents/StateComboBoxWithStyleClassSel.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    styleName="myCBStyle">
    <fx:Style>
        .myCBStyle {
            fontWeight : bold;
            fontSize : 15;
            }
    </fx:Style>
    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

***Note:** You cannot define a type selector in an MXML component. If you define a type selector, a compiler error occurs.*

Application developers can apply additional styles to the component. For example, if your component defines styles for the open duration and font size, application developers can still specify font color or other styles. The following example uses StateComboBoxWithStyleProps.mxml in an application and specifies the font color style for the control:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleWithPropsAddColor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <MyComp:StateComboBoxWithStyleProps color="red"/>

</s:Application>
```

Applying styles from the referencing file

When you reference an MXML component, the referencing file can specify style definitions to the MXML component by using the following mechanisms:

- Tag properties
- Class selectors
- Type selectors

The styles that application developers can apply correspond to the styles supported by the root tag of the MXML component. The following example uses a tag property to set a style for the custom MXML component:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MainStyleWithPropsOverrideOpenDur.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <MyComp:StateComboBoxWithStyleProps
        fontSize="12"/>
</s:Application>
```

When you specify styles as tag attributes, those styles override any conflicting styles set in the definition of the MXML component.

You can use a class selector to define styles. Often you use a class selector to apply styles to specific instances of a control, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MainStyleOverrideUsingClassSel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        .myStateComboBox {
            color : green;
        }
    </fx:Style>
    <MyComp:StateComboBoxWithStyleProps
        styleName="myStateComboBox"/>
    <s:ComboBox>
        <s:dataProvider>
            <s:ArrayList>
                <fx:String>AK</fx:String>
                <fx:String>AL</fx:String>
            </s:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

In this example, you use the `styleName` property in the tag definition of an MXML component to apply styles to a specific instance of the MXML component. However, those styles are not applied to the ComboBox control defined in the main application file, nor would they be applied to any other instances of StateComboBox.mxml unless you also specify the `styleName` property as part of defining those instances of the MXML component.

When you specify any styles by using a class selector, those styles override all styles that you set by using a class selector in the MXML file. Those styles do not override styles that you set by using tag properties in the MXML file.

You can also use a type selector to define styles. A type selector applies styles to all instances of a component, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MainStyleOverrideUsingTypeSel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <fx:Style>
        @namespace "myComponents.*";
        StateComboBoxWithStyleProps {
            color : red;
        }
    </fx:Style>
    <MyComp:StateComboBoxWithStyleProps/>
</s:Application>
```

In this example, the type selector specifies the `color` style for all instances of the StateComboBox control in the application. Notice that with type selectors, you also have to include a namespace definition for the component in the `<fx:Style>` tag.

When you specify any styles by using a type selector, those styles override all styles that you set by using a class selector in the MXML file. Those styles do not override styles that you set by using tag properties in the MXML file.

Applying a type selector to the root tag of a custom component

All custom components contain a root tag that specifies the superclass of the component. In the case of StateComboBox.mxml, the root tag is `<s:ComboBox>`. If you define a type selector for the ComboBox control, or for a superclass of the ComboBox control, in your main application file, that style definition is also applied to any custom component that uses a ComboBox control as its root tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/MainStyleOverrideUsingCBTypeSel.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|ComboBox {
            fontSize: 15;
            color: red;
        }
    </fx:Style>
    <MyComp:StateComboBoxWithStyleProps/>
    <s:ComboBox>
        <s:dataProvider>
            <s:ArrayList>
                <fx:String>AK</fx:String>
                <fx:String>AL</fx:String>
            </s:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

In this example, all ComboBox controls and all StateComboBox.mxml controls have a `fontSize` of 15 points and red text.

If you define a type selector for a superclass of the custom control and for the custom control itself, Flex ignores any conflicting settings from the type selector for the superclass, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleOverrideUsingCBTypeSelConflict.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace "myComponents.*";
        s|ComboBox {
            color: red;
            fontSize: 15;
        }
        StateComboBoxWithStyleProps {
            color: green;
        }
    </fx:Style>
    <MyComp:StateComboBoxWithStyleProps/>
    <s:ComboBox>
        <s:dataProvider>
            <s:ArrayList>
                <fx:String>AK</fx:String>
                <fx:String>AL</fx:String>
            </s:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

In this example, the StateComboBox control uses green text, and the values for the `fontSize` style specified in the type selector for the ComboBox control.

Applying styles from a defaults.css file

Flex includes a global style sheet, named defaults.css, inside the framework.swc file in the /frameworks/libs directory. This global style sheet contains style definitions for the global class selector and type selectors for most Flex components. Flex implicitly loads the defaults.css file and applies it to all Flex applications during compilation.

One of the most common ways for you to distribute your custom MXML and ActionScript components is to create a SWC file. A SWC file is an archive file of Flex components that make it easy to exchange components among Flex developers. You need to exchange only a single file, rather than the MXML or ActionScript files, images, and other resource files. The SWF file inside a SWC file is compiled, which means that the code is hidden from casual view.

For more information about SWC files, see "Flex compilers" on page 2164 and Build projects.

A SWC file can contain a local style sheet, named defaults.css, that contains the default style settings for the custom components defined within the SWC file. When Flex compiles your application, it automatically applies the local defaults.css to the components in your SWC file. In this way, you can override the global defaults.css style sheet with the local version in your SWC file.

The only requirements on the local version are:

- You can include only a single style sheet in the SWC file.
- The file must be named defaults.css.

- The file must be in the top-most directory of the SWC file.

For example, you define a custom ComboBox control, named StateComboBox.mxml, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_mxml/StateComboBox.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
            <!-- Add all other states. -->
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

Notice that this control does not define any style settings.

Then you create a defaults.css file in the same directory to define the default style settings for the custom component, as the following example shows:

```
@namespace "*";
StateComboBox
{
    arrowButtonWidth: 40;
    cornerRadius: 10;
    fontSize: "14";
    fontWeight: "bold";
    color: red;
    leading: 0;
    paddingLeft: 10;
    paddingRight: 10;
}
```

To create the SWC file, you run the compc command-line compiler from the directory that contains the defaults.css and StateComboBox.mxml files. Use the `include-file` option to specify the style sheet, as the following example shows:

```
flex_install_dir\bin\compc -source-path .
    -include-classes StateComboBox
    -include-file defaults.css defaults.css
    -o MyComponentsSWC.swc
```

This example creates a SWC file named MyComponentsSWC.swc.

***Note:** You can also use the `include-stylesheet` option to include the style sheet if the style sheet references assets that must be compiled, such as programmatic skins or other class files. For more information, see "Flex compilers" on page 2164.*

Then you write an application, named CreateCompsDefaultCSSApplication.mxml, that uses the StateComboBox control, as the following example shows:

```
<?xml version="1.0"?>
<!-- defaultCSS_app/CreateCompsDefaultCSSApplication.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:Label text="Custom MXML component using defaults.css from SWC file."/>
    <MyComp:StateComboBox/>

    <s:Label text="Default ComboBox control using the default defaults.css file."/>
    <s:ComboBox>
        <s:dataProvider>
            <s:ArrayList>
                <fx:String>AK</fx:String>
                <fx:String>AL</fx:String>
            </s:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

Notice that you include a namespace definition for the StateComboBox control in CreateCompsDefaultCSSApplication.mxml to reference the component. Because the MXML component is in the top-most directory of the SWC file, its package name is blank, and you reference it by using a namespace definition of `xmlns:MyComp="*"`.

To compile an application that uses MyComponentsSWC.swc, copy MyComponentsSWC.swc to the directory that contains the application. Then add the SWC file to the Library Path in Adobe® Flash® Builder™, or use the `-library-path` option to the mxmlc command-line compiler, as the following example shows:

```
flex_install_dir\bin\mxmlc
    -file-specs CreateCompsDefaultCSSApplication.mxml
    --library-path+=.
```

In the previous example, the defaults.css file and the component file were in the same directory. Typically, you place components in a directory structure, where the package name of the component reflects the directory location of the component in the SWC file.

In the next example, you put the StateComboBox.mxml file in the myComponents subdirectory of the SWC file, where the subdirectory corresponds to the package name of the component. However, defaults.css must still be in the top-level directory of the SWC file, regardless of the directory structure of the SWC file. You compile this SWC file by using the following command line:

```
flex_install_dir\bin\compc -source-path .
    -include-classes myComponents.StateComboBox
    -include-file defaults.css defaults.css
    -o MyComponentsSWC.swc
```

To use the StateComboBox.mxml component in your application, you define CreateCompsDefaultCSSApplication.mxml as the following example shows:

```
<?xml version="1.0"?>
<!-- defaultCSS_app/CreateCompsDefaultCSSApplicationSubDir.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:Label text="Custom MXML component using defaults.css from SWC file."/>
    <MyComp:StateComboBox/>

    <s:Label text="Default ComboBox control using the default defaults.css file."/>
    <s:ComboBox>
        <s:dataProvider>
            <s:ArrayList>
                <fx:String>AK</fx:String>
                <fx:String>AL</fx:String>
            </s:ArrayList>
        </s:dataProvider>
    </s:ComboBox>
</s:Application>
```

Notice that the namespace definition for the StateComboBox control in CreateCompsDefaultCSSApplication.mxml includes `myComponents.*` to match the directory structure, and package name, of the component in the SWC file.

You can modify and test your defaults.css file without having to recreate the SWC file by using the `-defaultscssfiles--` option to the compiler. The CSS files added by the `-defaults-css-files` option have a higher precedence than those in SWC files, so that they can override a corresponding definition in a SWC file.

When you are done modifying the defaults.css file, recreate the SWC file with the updated defaults.css file.

Advanced MXML components

One of the common goals of creating MXML components is to create configurable and reusable components. For example, you might want to create MXML components that take properties, dispatch events, define new style properties, have custom skins, or use other customizations.

For information about how to create and deploy simple MXML components, including how to apply styles and skins to your MXML components, see “Simple MXML components” on page 2399.

About reusable MXML components

One design consideration when you create custom MXML components is reusability. That is, do you want to create a component that is tightly coupled to your application, or one that is reusable in multiple applications?

A tightly coupled component is written for a specific application, and is often dependent on the application’s structure, variable names, or other details. If you change the application, you will probably need to modify the component to reflect that change, and it will also be difficult to use the component in another application without rewriting it.

Another possibility is to design a loosely coupled component for reuse. A loosely coupled component has a well-defined interface that specifies how to pass information to the component, and how the component passes back results to the application.

With loosely coupled components, you typically define properties of the component to let users pass information to it. These properties, defined by using variables or setter and getter methods, specify the data types of parameter values. For more information about defining component properties, see "Adding custom properties and methods to a component" on page 2413.

The best practice for defining components that return information to the main application is to design the component to dispatch an event that contains the return data. In that way, the main application can define an event listener to handle the event and take the appropriate action. For more information on dispatching events, see "Working with events" on page 2424.

Adding custom properties and methods to a component

MXML components provide you with a simple way to create ActionScript classes. When defining classes, you use class properties to store information and class methods to define class functionality. When creating MXML components, you can also add properties and methods to the components to make them configurable. By allowing the user to pass information to the components, you can create a reusable component that you can use in multiple locations throughout your application, or in multiple applications.

Defining properties and methods in MXML components

You can define methods for your MXML components in ActionScript, and properties in ActionScript or MXML. The Adobe Flex compiler converts the MXML component into an ActionScript class, so there is no performance difference between defining a property in MXML and defining it in ActionScript.

Defining properties and methods in ActionScript

With ActionScript, you define properties and methods by using the same syntax that you use in an ActionScript class. For more information on using ActionScript to define properties and methods, see "Custom ActionScript components" on page 2363.

When using ActionScript, you place a property or method definition within an `<fx:Script>` block. The `<fx:Script>` tag must be an immediate child tag of the root tag of the MXML file. A public variable declaration or a set function in an `<fx:Script>` tag becomes a property of the component. A public ActionScript function in an `<fx:Script>` tag becomes a method of the component.

In the following example, the component defines two data providers to populate the ComboBox control, and a function to use as the event listener for the `creationComplete` event. This function sets the data provider of the ComboBox based on the value of the `shortNames` property. By default, the `shortNames` property is set to `true`, to display two-letter names.

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPropAS.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="setNameLength();">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            // Define public variables.
            public var shortNames:Boolean = true;

            // Define private variables.
            private var stateArrayShort:ArrayList = new ArrayList(["AK", "AL"]);
            private var stateArrayLong:ArrayList = new ArrayList(["Arkansas", "Alaska"]);
            // Define listener method.
            public function setNameLength():void {
                if (shortNames) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
            }
        ]]>
    </fx:Script>
</s:ComboBox>
```

The following MXML application file uses the `<MyComp:StateComboBoxPropAS>` tag to configure the control to display long state names:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <MyComp:StateComboBoxPropAS shortNames="true"/>

</s:Application>
```

The following example modifies the component to add a method that lets you change the display of the state name at run time. This public method takes a single argument that specifies the value of the `shortNames` property.

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPropMethod.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="setNameLength();">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            // Define public variables.
            public var shortNames:Boolean = true;

            // Define private variables.
            private var stateArrayShort:ArrayList = new ArrayList(["AK", "AL"]);
            private var stateArrayLong:ArrayList = new ArrayList(["Arkansas", "Alaska"]);
            public function setNameLength():void {
                if (shortNames) {
                    this.dataProvider=stateArrayShort; }
                else {
                    this.dataProvider=stateArrayLong; }
            }

            public function setShortName(val:Boolean):void {
                shortNames=val;
                if (val) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
            }
        ]]>
    </fx:Script>
</s:ComboBox>
```

You might use this new method with the `click` event of a Button control to change the display from long names to short names, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropWithMethod.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <MyComp:StateComboBoxPropMethod id="myCB"
        shortNames="false"/>
    <s:Button label="Use Short Names"
        click="myCB.setShortName(true);"/>
</s:Application>
```

Defining properties in MXML

In MXML, you can use an MXML tag to define a property of any type, as long as the type refers to an ActionScript class name. For example, you can use the `<fx:String>`, `<fx:Number>`, and `<fx:Boolean>` tags to define properties in your MXML components that take String, Number, or Boolean values, respectively. When using one of these tags, you must specify an `id`, which becomes the property name.

Optionally, you can specify an initial value in the body of the tag, or you can use the `source` property to specify the contents of an external URL or file as the initial property value. If you use the `source` property, the body of the tag must be empty. The initial value can be static data or a binding expression.

The following examples show initial properties set as static data and binding expressions. Values are set in the tag bodies and in the `source` properties.

```
<!-- Boolean property examples: -->
<fx:Boolean id="myBooleanProperty">true</fx:Boolean>
<fx:Boolean id="passwordStatus">{passwordExpired}</fx:Boolean>

<!-- Number property examples: -->
<fx:Number id="myNumberProperty">15</fx:Number>
<fx:Number id="minutes">{numHours * 60}</fx:Number>

<!-- String property examples: -->
<fx:String id="myStringProperty">Welcome, {CustomerName}.</fx:String>
<fx:String id="myStringProperty1" source="./file"/>
```

All properties defined by using the `<fx:String>`, `<fx:Number>`, and `<fx:Boolean>` tags are public. This means that the component user can access these properties.

The following example modifies the example in "Defining properties and methods in ActionScript" on page 2413 to define the `shortNames` property by using an MXML tag, rather than an ActionScript variable definition:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPropMXML.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="setNameLength();">
    <fx:Declarations>
        <!-- Control display of state names. -->
        <fx:Boolean id="shortNames">true</fx:Boolean>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            // Define private variables.
            private var stateArrayShort:ArrayList = new ArrayList(["AK", "AL"]);
            private var stateArrayLong:ArrayList = new ArrayList(["Arkansas", "Alaska"]);

            // Define listener method.
            public function setNameLength():void {
                if (shortNames) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
            }
        ]]>
    </fx:Script>
</s:ComboBox>
```

In the preceding example, you implement the StateComboBox.mxml file by using the `<fx:Boolean>` tag to add a new property, `shortNames`, with a default value of `true`. This property controls whether the ComboBox control displays state names that use a two-letter format, or the entire state name.

Defining properties by using setters and getters

You can define properties for your MXML components by using setter and getter methods. The advantage of getters and setters is that they isolate the variable from direct public access so that you can perform the following tasks:

- Inspect and validate any data written to the property on a write operation
- Trigger events that are associated with the property when the property changes
- Calculate a return value on a read operation

For more information, see "Custom ActionScript components" on page 2363.

In the following example, the StateComboBoxGetSet.mxml component contains several new properties and methods:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxSetGet.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;

            // Define private variables.
            private var stateArrayShort:ArrayList = new ArrayList(["AK", "AL"]);
            private var stateArrayLong:ArrayList = new ArrayList(["Arkansas", "Alaska"]);
            // Variable holding the display setting.
            private var __shortNames:Boolean = true;

            // Set method.
            public function set shortNames(val:Boolean):void {
                // Call method to set the dataProvider
                // based on the name length.
                __shortNames = val;
                if (__shortNames) {
                    this.dataProvider=stateArrayShort; }
                else {
                    this.dataProvider=stateArrayLong; }
            }
            // Get method.
            public function get shortNames():Boolean{
                return __shortNames;
            }
        ]]>
    </fx:Script>
</s:ComboBox>
```

In this example, you create a StateComboBoxGetSet.mxml control that takes a `shortNames` property defined by using ActionScript setter and getter methods. One advantage to using setter and getter methods to define a property is that the component can recognize changes to the property at run time. For example, you can give your users the option of displaying short state names or long state names from the application. The setter method modifies the component at run time in response to the user's selection.

You can also define events to be dispatched when a property changes. This enables you to signal the change so that an event listener can recognize the change. For more information on events, see "Working with events" on page 2424.

You can call a component's custom methods and access its properties in ActionScript just as you would any instance method or component property, as the following application shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropSetGet.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <MyComp:StateComboBoxSetGet id="myStateCB"
        shortNames="true"/>
    <s:Button label="Toggle Short Names"
        click="myStateCB.shortNames=!myStateCB.shortNames;"/>
</s:Application>
```

In this example, selecting the button toggles the display format of the state name between the short and long formats.

Defining inspectable properties

You should precede the variable or set function with the `[Inspectable]` metadata tag if you plan to use the component in an authoring tool such as Adobe Flash Builder™. The `[Inspectable]` metadata tag must immediately precede the property's variable declaration or the setter and getter methods to be bound to that property, as the following example shows:

```
<fx:Script>
    <![CDATA[

        // Define public variables.
        [Inspectable(defaultValue=true)]
        public var shortNames:Boolean = true;

    ]]>
</fx:Script>
```

For more information on the `[Inspectable]` metadata tag, see "Metadata tags in custom components" on page 2376.

Supporting data binding in custom properties

The Flex data binding mechanism provides a syntax for automatically copying the value of a property of one object to a property of another object at run time. The following example shows a Text control that gets its data from Slider control's `value` property. The property name inside the curly braces ({ }) is a binding expression that copies the value of the source property, `mySlider.value`, to the destination property, the Text control's `text` property, as the following example shows:

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

Data binding is usually triggered whenever the value of the source property changes.

Properties that you define in your custom controls can also take advantage of data binding. You can automatically use any property defined by using an MXML tag, such as `<fx:Boolean>`, and any ActionScript property defined as a variable or defined by using setter and getter methods as the destination of a binding expression.

For example, "Defining properties by using setters and getters" on page 2417 defined the `shortNames` property of StateComboBoxGetSet.mxml by using setter and getter methods. With no modification to that component, you can use `shortNames` as the destination of a binding expression, as the following example shows:

```
<MyComp:StateComboBoxSetGet shortNames="{some_prop}"/>
```

However, you can also write your component to use the `shortNames` property as the source of a binding expression, as the following example shows for the component StateComboBoxGetSetBinding.mxml:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropSetGetBinding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <MyComp:StateComboBoxSetGetBinding id="myStateCB"
        shortNames="false"/>

    <s:TextArea
        text="The value of shortNames is {myStateCB.shortNames}"/>
    <s:Button label="Toggle Short Names"
        click="myStateCB.shortNames=!myStateCB.shortNames;"/>
</s:Application>
```

When a property is the source of a data binding expression, any changes to the property must signal an update to the destination property. The way to signal that change is to dispatch an event, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxSetGetBinding.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayList;
            import flash.events.Event;
            // Define private variables.
            private var stateArrayShort:ArrayList = new ArrayList(["AK", "AL"]);
            private var stateArrayLong:ArrayList = new ArrayList(["Arkansas", "Alaska"]);
            private var __shortNames:Boolean = true;

            public function set shortNames(val:Boolean):void {
                __shortNames = val;
                if (__shortNames) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
                    // Create and dispatch event.
                    dispatchEvent(new Event("changeShortNames"));
            }
            // Include the [Bindable] metadata tag.
            [Bindable(event="changeShortNames")]
            public function get shortNames():Boolean {
                return __shortNames;
            }
        ]]>
    </fx:Script>
</s:ComboBox>
```

Use a property as the source of a data binding expression

1 Define the property as a variable, or by using setter and getter methods.

You must define a setter method and a getter method if you use the `[Bindable]` tag with the property.

2 Insert the `[Bindable]` metadata tag before the property definition, or before either the setter or getter method, and optionally specify the name of the event dispatched by the property when it changes.

If you omit the event name specification from the `[Bindable]` metadata tag, Flex automatically generates and dispatches an event named `propertyChange`. If the property value remains the same on a write, Flex does not dispatch the event or update the property.

Alternatively, you can place the `[Bindable]` metadata before a public class definition. This makes all public properties that you defined as variables, and all public properties that you defined by using both a setter and a getter method, usable as the source of a binding expression.

***Note:** When you use the `[Bindable]` metadata tag before a public class definition, it only applies to public properties; it does not apply to private or protected properties, or to properties defined in any other namespace. You must insert the `[Bindable]` metadata tag before a nonpublic property to make it usable as the source for a data binding expression.*

3 Add a call to the `dispatchEvent()` method to dispatch the event when you define the event name in the `[Bindable]` metadata tag.

For more information on using the `[Bindable]` tag, see “Bindable metadata tag” on page 2381.

Passing references to properties of MXML components

One of the ways that you can make a component reusable is to design it so that users can pass values to the component by using public properties of the component. For information on how to define properties for MXML components by using MXML and ActionScript, and how to pass values to those properties, see “Supporting data binding in custom properties” on page 2418.

Rather than passing a value to a component, you can pass a reference to it. The reference could be to the calling component, to another component, or to a property of a component.

Accessing the Application object

The Application object is the top-level object in a Flex application. Often, you must reference properties or objects of the Application object from your custom component. Use the mx.core.FlexGlobals.topLevelApplication static property to reference the application object.

You can also use the parentDocument property to reference the next object up in the document chain of a Flex application. The `parentDocument` property is inherited by all components from the UIComponent class. For an MXML component, the `parentDocument` property references the Object corresponding to the component that referenced the MXML component.

For more information on the `mx.core.FlexGlobals.topLevelApplication` static property and the `parentDocument` property, see “Application containers” on page 393.

Even if the calling file does not pass a reference to the Application object, you can always access it from your MXML component. For example, the following application contains a custom component called StateComboBoxDirectRef. In this example, StateComboBoxDirectRef is designed to write the index of the selected item in the ComboBox to the TextArea control:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainDirectRef.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="myTAMain"/>
    <MyComp:StateComboBoxDirectRef/>
</s:Application>
```

The simplest way to write StateComboBoxDirectRef.mxml is to use the `mx.core.FlexGlobals.topLevelApplication` static property to write the index directly to the TextArea control, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxDirectRef.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    close="handleCloseEvent(event);">
    <fx:Script>
        <![CDATA[

            import flash.events.Event;
            import mx.core.FlexGlobals;

            public function handleCloseEvent(eventObj:Event):void {
                mx.core.FlexGlobals.topLevelApplication.myTAMain.text=
                    String(this.selectedIndex);
            }
        ]]>
    </fx:Script>

    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

In the previous example, you use the `close` event of the ComboBox control to write the `selectedIndex` directly to the TextArea control in the main application. You must cast the value of `selectedIndex` to a String because the `text` property of the TextArea control is of type String.

You could make the custom component slightly more reusable by using the `parentDocument` property to reference the TextArea control, rather than the `mx.core.FlexGlobals.topLevelApplication` static property. By using the `parentDocument` property, you can call the custom component from any other MXML component that contains a TextArea control named myTAMain, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxDirectRefParentObj.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    close="handleCloseEvent(event);">
    <fx:Script>
        <![CDATA[

            import flash.events.Event;

            public function handleCloseEvent(eventObj:Event):void {
                parentDocument.myTAMain.text=String(selectedIndex);
            }
        ]]>
    </fx:Script>

    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

Although these examples work, they require that the TextArea control has a predefined `id` property, and that MXML component knows that `id`. In this case, the custom component is an example of a tightly coupled component. That is, the component is written for a specific application and application structure, and it is not easily reused in another application.

Passing a reference to the component

A loosely coupled component is a highly reusable component that you can easily use in different places in one application, or in different applications. To make the component from "Supporting data binding in custom properties" on page 2418 reusable, you can pass a reference to the TextArea control to the custom component, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPassRefToTA.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="myTAMain" />
    <MyComp:StateComboBoxPassRefToTA outputTA="{myTAMain}" />
</s:Application>
```

The custom component does not have to know anything about the main application, other than that it writes its results back to a TextArea control, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPassRefToTA.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    close="handleCloseEvent(event);">
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            import spark.components.TextArea;

            // Define a variable of type mx.controls.TextArea.
            public var outputTA:TextArea;

            public function handleCloseEvent(eventObj:Event):void {
                outputTA.text=String(this.selectedIndex);
            }
        ]]>
    </fx:Script>

    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

In this example, you use the Flex data binding syntax to pass the reference to the TextArea control to your custom component. Now, you can use StateComboBoxPassRefToTA.mxml anywhere in an application. The only requirement is that the calling component must pass a reference to a TextArea control to the component.

Passing a reference to the calling component

In “Passing a reference to the component” on page 2422, you passed a reference to a single component to the custom MXML component. This allowed the MXML component to access only a single component in the main application.

One type of reference that you can pass to your component is a reference to the calling component. With a reference to the calling component, your custom MXML file can access any properties or object in the calling component.

To pass a reference to the calling component to a custom MXML component, you create a property in the custom MXML component to represent the calling component. Then, from the calling component, you pass the reference, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/CallingComponent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout horizontalAlign="left"/>
    </s:layout>
    <!-- Use the caller property to pass a reference to the
        calling component to DestinationComp. -->
    <s:Label text="Enter text"/>
    <s:TextInput id="text1" text="Hello"/>
    <s:Label text="Input text automatically copied to MXML component."/>
    <MyComp:DestinationComp caller="{this}"/>
</s:Application>
```

In the definition of DestinationComp.mxml, you define the `caller` property, and specify as its data type the name of the file of the calling MXML file, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/DestinationComp.mxml -->
<s:VGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            // Define variable to reference calling file.
            [Bindable]
            public var caller:CallingComponent;
        ]]>
    </fx:Script>

    <s:TextInput id="mytext"
        text="{caller.text1.text}"/>
</s:VGroup>
```

Remember, an MXML component corresponds to an ActionScript class, where the ActionScript class name is the filename of the MXML component. Therefore, the MXML component defines a new data type. You can then create a variable whose data type is that of the calling file.

With the reference to the calling file, your MXML component can access any property of the calling file, and you can bind the value of the TextInput control in CallingComp.mxml to the TextInput control in StateComboBox.mxml. Creating a property of type CallingComp provides strong typing benefits and ensures that binding works correctly.

Working with events

Flex applications are event-driven. Events let a programmer know when the user interacts with the interface, and also when important changes happen in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing. You can handle events that your custom components generate and add your own event types to your custom components.

Handling events from simple MXML components

Simple MXML components are those that contain a single root tag that is not a container. In this topic, the StateComboBox.mxml component is a simple component because it contains a definition only for the ComboBox control.

You have two choices for handling events that a simple component dispatches: handle the events within the definition of your MXML component, or allow the file that references the component to handle them.

The following example uses the StateComboBox.mxml component, and defines the event listener for the component's `close` event in the main application:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="*">

    <fx:Script>
        <![CDATA[
            import flash.events.Event;

            public function handleCloseEvent(eventObj:Event):void {
                ...
            }
        ]]>
    </fx:Script>

    <MyComp:StateComboBox rowCount="5" close="handleCloseEvent(event);"/>

</s:Application>
```

In this example, if the MXML component dispatches a `close` event, the event listener in the calling MXML file handles it.

Alternatively, you could define the event listener within the StateComboBox.mxml component, as the following example shows:

```
<?xml version="1.0"?>
<!-- StateComboBox.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    close="handleCloseEvent(event);">

    <fx:Script>
        <![CDATA[

            import flash.events.Event;

            public function handleCloseEvent(eventObj:Event):void {
                ...
            }
        ]]>
    </fx:Script>

    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

With simple MXML components, you can define event listeners in both places, and both event listeners process the event. However, the event listeners defined within the component execute before any listeners defined in the application.

Creating custom events

All MXML components can dispatch events, either those inherited by the components from their superclasses, or new events that you define within your components. When you are developing MXML components, you can add your own event types.

In this example, you define a new component called TextAreaEnabled.mxml that uses a `<s:TextArea>` tag as its root tag. This component also defines a new property called `enableTA` that users set to `true` to enable text input or to `false` to disable input.

The setter method dispatches a new event type, called `enableChanged`, when the value of the `enableTA` variable changes. The `[Event]` metadata tag identifies the event to the MXML compiler so that the file referencing the component can use the new property. For more information on using the `[Event]` metadata keyword, see “Metadata tags in custom components” on page 2376.

The syntax for the `[Event]` metadata tag is as follows:

```
<fx:Metadata>
    [Event(name="eventName", type="eventType")]
</fx:Metadata>
```

You dispatch new event types by using the `dispatchEvent()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/TextAreaEnabled.mxml -->
<s:TextArea xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
    <fx:Metadata>
        [Event(name="enableChanged", type="flash.events.Event")]
    </fx:Metadata>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            // Define private variable to hold the enabled state.
            private var __enableTA:Boolean;
            // Define a setter method for the private variable.
            public function set enableTA(val:Boolean):void {
                __enableTA = val;
                enabled = val;

                // Define event object, initialize it, then dispatch it.
                dispatchEvent(new Event("enableChanged"));
            }
            // Define a getter method for the private variable.
            public function get enableTA():Boolean {
               return __enableTA;
            }
        ]]>
    </fx:Script>
</s:TextArea>
```

The following main application includes TextAreaEnabled.mxml and defines an event listener for the `enableChanged` event:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainTextAreaEnable.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            import myComponents.TextAreaEnabled;

            public function handleEnableChangeEvent(eventObj:Event):void {
                var tempTA:TextAreaEnabled =
                    eventObj.currentTarget as TextAreaEnabled;
                if (tempTA.enableTA) {
                    myButton.label="Click to disable";
                    }
                else {
                    myButton.label="Click to enable";
                    }
            }
        ]]>
    </fx:Script>
    <MyComp:TextAreaEnabled id="myTA" enableTA="false"
        enableChanged="handleEnableChangeEvent(event);" />

    <s:Button id="myButton" label="Click to enable"
        click="myTA.enableTA=!myTA.enableTA;" />
</s:Application>
```

If you do not use the `[Event]` metadata tag in the custom component file to define the `enableChanged` event, the MXML compiler generates an error message when you reference the event name in an MXML file. Any component can register an event listener for the event in ActionScript using the `addEventListener()` method, even if you omit the `[Event]` metadata tag.

You can also create and dispatch events that use an event object of a type other than that defined by the Event class. For example, you might want to create an event object that contains new properties so that you can pass those properties back to the referencing file. To do so, you create a subclass of the Event class to define your new event object. For information on creating custom event classes, see “Custom events” on page 2369.

Handling events from composite components

Composite components are components that use a container for the root tag, and define child components in that container. You handle events generated by the root container in the same way as you handle events generated by simple MXML components. That is, you can handle the event within the MXML component, within the referencing file, or both. For more information, see “Handling events from simple MXML components” on page 2425.

To handle an event that a child of the root container dispatches, you can handle it in the MXML component in the same way as you handle an event from the root container. However, if a child component of the root container dispatches an event, and you want that event to be dispatched to the referencing file, you must add logic to your custom component to propagate the event.

For example, you can define a component that uses an `<mx:Form>` tag as the root tag, and include within it a ComboBox control. Any event that the Form container dispatches, such a `scroll` event, is dispatched to the referencing file of the custom component. However, the `close` event of the ComboBox control is dispatched only within the custom MXML component.

To propagate the `close` event outside of the custom component, you define an event listener for it in the MXML component that redispatches it, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/AddressForm.mxml -->
<s:Form xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:local="*">
    <fx:Metadata>
        [Event(name="close", type="flash.events.Event")]
    </fx:Metadata>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            // Redispatch event.
            private function handleCloseEventInternal(eventObj:Event):void {
                dispatchEvent(eventObj);
            }
        ]]>
    </fx:Script>
    <s:FormItem label="Name">
        <s:TextInput id="name1" />
    </s:FormItem>
    <s:FormItem label="Street">
        <s:TextInput id="street" />
    </s:FormItem>
    <s:FormItem label="City" >
        <s:TextInput id="city" />
    </s:FormItem>
    <s:FormItem label="State" >
        <s:ComboBox close="handleCloseEventInternal(event);">
            <s:dataProvider>
                <s:ArrayList>
                    <fx:String>AK</fx:String>
                    <fx:String>AL</fx:String>
                </s:ArrayList>
            </s:dataProvider>
        </s:ComboBox>
    </s:FormItem>
</s:Form>
```

In this example, you propagate the event to the calling file. You could, alternatively, create an event type and new event object as part the propagation. For more information on the `[Event]` metadata tag, see “Metadata tags in custom components” on page 2376.

You can handle the `close` event in your main application, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainAddressFormHandleEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*"
    height="600">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[

            import flash.events.Event;

            private function handleCloseEvent(eventObj:Event):void {
                myTAClose.text=eventObj.type;
            }
            private function handleMouseDown(eventObj:Event):void {
                myTA.text=eventObj.type;
            }
        ]]>
    </fx:Script>
    <s:TextArea id="myTA" />
    <s:TextArea id="myTAClose" />
    <MyComp:AddressForm mouseDown="handleMouseDown(event);"
        close="handleCloseEvent(event);"/>
</s:Application>
```

About interfaces

Interfaces are a type of class that you design to act as an outline for your components. When you write an interface, you provide only the names of public methods rather than any implementation. For example, if you define two methods in an interface and then implement that interface, the implementing class must provide implementations of those two methods.

Interfaces in ActionScript can declare methods and properties only by using setter and getter methods; they cannot specify constants. The benefit of interfaces is that you can define a contract that all classes that implement that interface must follow. Also, if your class implements an interface, instances of that class can also be cast to that interface.

Custom MXML components can implement interfaces just as other ActionScript classes can. To do this, you use the `implements` attribute. All MXML tags support this attribute.

The following code is an example of a simple interface that declares several methods:

```
// The following is in a file named SuperBox.as.
interface SuperBox {
    function selectSuperItem():String;
    function removeSuperItem():Boolean;
    function addSuperItem():Boolean;
}
```

A class that implements the SuperBox interface uses the `implements` attribute to point to its interface and must provide an implementation of the methods. The following example of a custom ComboBox component implements the SuperBox interface:

```
<?xml version="1.0"?>
<!-- StateComboBox.mxml -->
<s:ComboBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    implements="SuperBox">
    <fx:Script>
        <![CDATA[
            public function selectSuperItem():String {
                return "Super Item was selected";
            }
            public function removeSuperItem():Boolean {
                return true;
            }
            public function addSuperItem():Boolean {
                return true;
            }
        ]]>
    </fx:Script>
    <s:dataProvider>
        <s:ArrayList>
            <fx:String>AK</fx:String>
            <fx:String>AL</fx:String>
        </s:ArrayList>
    </s:dataProvider>
</s:ComboBox>
```

You can implement multiple interfaces by separating them with commas, as the following example shows:

```
<s:ComboBox ... implements="SuperBox, SuperBorder, SuperData">
```

All methods that you declare in an interface are considered public. If you define an interface and then implement that interface, but do not implement all of its methods, the MXML compiler throws an error.

Methods that are implemented in the custom component must have the same return type as their corresponding methods in the interface. If no return type is specified in the interface, the implementing methods can declare any return type.

About implementing IMXMLObject

You cannot define a constructor for an MXML component. If you do, the Flex compiler issues an error message that specifies that you defined a duplicate function.

For many types of Flex components, you can use an event listener instead of a constructor. For example, depending on what you want to do, you can write an event listener for the `preinitialize`, `initialize`, or `creationComplete` event to replace the constructor.

These events are all defined by the UIComponent class, and inherited by all of its subclasses. If you create an MXML component that is not a subclass of UIComponent, you cannot take advantage of these events. You can instead implement the IMXMLObject interface in your MXML component, and then implement the `IMXMLObject.initialized()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/ObjectComp.mxml -->
<fx:Object xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    implements="mx.core.IMXMLObject">
    <fx:Script>
        <![CDATA[

            // Implement the IMXMLObject.initialized() method.
            public function initialized(document:Object, id:String):void {
                trace("initialized, x = " + x);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <fx:Number id="y"/>
        <fx:Number id="z"/>
        <fx:Number id="x"/>
    </fx:Declarations>
</fx:Object>
```

Flex calls the `IMXMLObject.initialized()` method after it initializes the properties of the component. The following example uses this component:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainInitObject.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*"
    creationComplete="initApp();">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            public function initApp():void {
                myTA.text="myFC.x = " + String(myFC.x);
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <MyComp:ObjectComp id="myFC" x="1" y="2" z="3"/>
    </fx:Declarations>
    <s:TextArea id="myTA"/>
</s:Application>
```

Because Flex calls the `IMXMLObject.initialized()` method after it initializes the properties of the component, the `trace()` function in the implementation of the `IMXMLObject.initialized()` method outputs the following:

```
initialized, x = 1
```

Create simple visual components in ActionScript

You define custom ActionScript components to extend the Adobe® Flex® component library. For example, you can create a customized Button, Tree, or DataGrid component as an ActionScript component.

For information on creating advanced components in ActionScript, see “Create advanced Spark visual components in ActionScript” on page 2451 and “Create advanced MX visual components in ActionScript” on page 2475.

About ActionScript components

You create reusable components by using ActionScript, and reference these components in your Flex applications as MXML tags. Components created in ActionScript can contain graphical elements, define custom business logic, or extend existing Flex components.

Flex components are implemented as a class hierarchy in ActionScript. Each component in your application is an instance of an ActionScript class. All Flex visual components are derived from the ActionScript UIComponent class. To create your own components, you can create a subclass from the UIComponent class or from any of its subclasses. For a complete description of the class hierarchy, see the ActionScript 3.0 Reference for the Adobe Flash Platform.

The class you choose to use as the superclass of your custom component depends on what you are trying to accomplish. For example, you might require a custom button control. You could create a subclass of the UIComponent class, and then recreate all of the functionality built into the Flex Button class. A better and faster way to create your custom button component is to create a subclass of the Flex Button class, and then modify it in your custom class.

Simple components are subclasses of existing Flex components that modify the behavior of the component, or add new functionality to it. For example, you might add a new event type to a Button control, or modify the default styles or skins of a DataGrid control.

You can also create advanced ActionScript components. Advanced ActionScript components might have one of the following requirements:

- Modify the appearance of a control or the layout functionality of a container
- Encapsulate one or more components into a composite component
- Subclass UIComponent to create components

For information on creating advanced ActionScript components, see “Create advanced MX visual components in ActionScript” on page 2475.

Example: Creating a simple component

When you define a simple component, you do not create a component yourself, but instead modify the behavior of an existing component. In this section, you create a customized TextArea control by extending the spark.components.TextArea component. This component adds an event listener for the keyDown event to the TextArea control. The `KeyDown` event deletes all the text in the control when a user presses the Control+Alt+Z key combination.

```
package myComponents
{
    // createcomps_as/myComponents/DeleteTextArea.as
    import spark.components.TextArea;
    import flash.events.KeyboardEvent;

    public class DeleteTextArea extends TextArea {

        // Constructor
        public function DeleteTextArea() {
            // Call super().
            super();
            // Add event listener for keyDown event.
            addEventListener("keyDown", myKeyDown);
        }
        // Define private keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Check to see if Ctrl+Alt+Z pressed. Keycode is 90.
            if (eventObj.ctrlKey && eventObj.keyCode == 90)
                text = "";
        }
    }
}
```

The filename for this component is DeleteTextArea.as, and its location is the myComponents subdirectory of the application, as specified by the `package` statement. For more information on using the `package` statement and specifying the directory location of your components, see "Custom ActionScript components" on page 2363.

You can now use your new TextArea control in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainDeleteTextArea.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <MyComp:DeleteTextArea/>
</s:Application>
```

Note: *Your class must be specified as* `public` *for you to be able to access it by using an MXML tag.*

In this example, you first define the `MyComp` namespace to specify the location of your custom component. You then reference the component as an MXML tag by using the namespace prefix.

You can specify any inherited properties of the superclass in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainDeleteTextAreaProps.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <MyComp:DeleteTextArea
        maxChars="50"
        text="My Message"/>
</s:Application>
```

You do not have to change the name of your custom component when you create a subclass of a Flex class. In the previous example, you could have named your custom component TextArea, and written it to the TextArea.as file in the myComponents directory, as the following example shows:

```
package myComponents
{
    import spark.components.TextArea;
    import flash.events.KeyboardEvent;

    public class TextArea extends TextArea {
        ...
    }
}
```

You can now use your custom TextArea control, and the standard TextArea control, in an application. To differentiate between the two controls, you use the namespace prefix, as the following example shows:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*" >
    <MyComp:TextArea/>
    <s:TextArea/>
</s:Application>
```

Adding properties and methods to a component

To make your custom components reusable, you design them so that users can pass information to them. You do this by adding public properties and methods to your components, and by making the components accessible in MXML.

Defining public properties in ActionScript

You can use one of the following methods to add public properties to your ActionScript components:

- Define public variables
- Define public getter and setter methods

Accessing public properties in MXML

All public properties defined in your component are accessible in MXML by using MXML tag properties. For example, you might allow the user to pass a value to your component, as the following example shows:

```
<MyComp:MyCustomComponent prop1="3"/>
```

To create a component that takes tag attributes in MXML, you define a public variable with the same name as the tag attribute in your class definition:

```
public class MyCustomComponent extends TextArea {

    // Define an uninitialized variable.
    public var prop1:Number;

    // Define and initialize a variable.
    public var prop2:Number=5;
    ...
}
```

You can also use public getter and setter methods to define a property, as the following example shows:

```
public class MyCustomComponent extends TextArea {

    private var _prop1:Number;

    public function get prop1():Number {
        // Method body.
        // Typically the last line returns the value of the private variable.
        return _prop1;
    }

    public function set prop1(value:Number):void {
        // Typically sets the private variable to the argument.
        _prop1=value;
        // Define any other logic, such as dispatching an event.
    }
}
```

You can define and initialize a private variable, as the following example shows:

```
private var _prop2:Number=5;
```

When you specify a value to the property in MXML, Flex automatically calls the setter method. If you do not set the property in MXML, Flex sets it to its initial value, if you specified one, or to the type's default value, which is `NaN` for a variable of type Number.

Defining public properties as variables

In the following example, you use the Control+Alt+I key combination to extend the TextArea control to let the user increase the font size by one point, or use the Control+Alt+M key combination to decrease the font size by one point:

```
package myComponents
{
    // createcomps_as/myComponents/TextAreaFontControl.as
    import spark.components.TextArea;
    import flash.events.KeyboardEvent;
    import flash.events.Event;

    public class TextAreaFontControl extends TextArea
    {
        // Constructor
        public function TextAreaFontControl() {
            super();
            // Add event listeners.
            addEventListener("keyDown", myKeyDown);
            addEventListener("creationComplete", myCreationComplete);
        }
        // Define private var for current font size.
        private var currentFontSize:Number;
        // Define a public property for the minimum font size.
        public var minFontSize:Number = 5;
        // Define a public property for the maximum font size.
        public var maxFontSize:Number = 15;

        // Initialization event handler for getting default font size.
        private function myCreationComplete(eventObj:Event):void {
            // Get current font size
```

```
             currentFontSize = getStyle('fontSize');
        }
        // keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Was Ctrl key pressed?
            if (eventObj.ctrlKey)
            {
                switch (eventObj.keyCode) {
                    // Was Ctrl+Alt+I pressed?
                    case 73 :
                        if (currentFontSize < maxFontSize) {
                            currentFontSize = currentFontSize + 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    // Was Ctrl+Alt+ pressed?
                    case 77 :
                        if (currentFontSize > minFontSize) {
                            currentFontSize = currentFontSize - 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    default :
                        break;
                }
            }
        }
    }
}
```

Notice that the call to the `getStyle()` method is in the event listener for the `creationComplete` event. You must wait until component creation is complete before calling `getStyle()` to ensure that Flex has set all inherited styles. However, you can call `setStyle()` in the component constructor to set styles.

This example uses variables to define public properties to control the maximum font size, `maxFontSize`, and minimum font size, `minFontSize`, of the control. Users can set these properties in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainTextAreaFontControl.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <MyComp:TextAreaFontControl id="myTAFS"
        minFontSize="8"
        maxFontSize="50"/>

    <s:Button
        label="Get Font Size"
        click="myTA.text=String(myTAFS.getStyle('fontSize'));"/>
    <s:TextArea id="myTA"/>
</s:Application>
```

Defining public properties by using getter and setter methods

There are no restrictions on using public variables to define public properties, However, Adobe recommends that you use getter and setter methods so that you can control user interaction with your component, as described in “Defining properties as getters and setters” on page 2367.

The following example code defines a component named TextAreaFontControlGetSet that replaces the public property definition for the `maxFontSize` property shown in “Defining public properties as variables” on page 2436:

```
package myComponents
{
    // createcomps_as/myComponents/TextAreaFontControlGetSet.as
    import spark.components.TextArea;
    import flash.events.KeyboardEvent;
    import flash.events.Event;

    public class TextAreaFontControlGetSet extends TextArea
    {
        public function TextAreaFontControlGetSet()
        {
            super();
            addEventListener("keyDown", myKeyDown);
            addEventListener("creationComplete", myCreationComplete);
        }
        private var currentFontSize:Number;
        public var minFontSize:Number = 5;
        // Define private variable for maxFontSize.
        private var _maxFontSize:Number = 15;

        // Define public getter method.
        public function get maxFontSize():Number {
            return _maxFontSize;
        }

        // Define public setter method.
        public function set maxFontSize(value:Number):void {
            if (value <= 30) {
                _maxFontSize = value;
            } else _maxFontSize = 30;
        }

        private function myCreationComplete(eventObj:Event):void {
            // Get current font size
             currentFontSize = getStyle('fontSize');
        }
        // keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Was Ctrl key pressed?
            if (eventObj.ctrlKey)
            {
                switch (eventObj.keyCode) {
                    // Was Ctrl+Alt+I pressed?
                    case 73 :
                        if (currentFontSize < maxFontSize) {
                            currentFontSize = currentFontSize + 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    // Was Ctrl+Alt+M pressed?
                    case 77 :
                        if (currentFontSize > minFontSize) {
                            currentFontSize = currentFontSize - 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    default :
                        break;
                }
            }
        }
    }
}
```

In this example, the setter method checks that the specified font size is less than the predefined limit of 30 pixels. If the font size is greater than the limit, it sets it to the limit.

Creating a default property

You can define a default property for your ActionScript components by using the `[DefaultProperty]` metadata tag. You can then use the default property in MXML as the child tag of the component tag without specifying the property name. For more information on using the default property, including an example, see "Defining public properties as variables" on page 2436.

You can use the `[DefaultProperty]` metadata tag in your ActionScript component to define a single default property, as the following example shows:

```
package myComponents
{
    // createcomps_as/myComponents/TextAreaDefaultProp.as
    import spark.components.TextArea;
    // Define the default property.
    [DefaultProperty("defaultText")]
    public class TextAreaDefaultProp extends TextArea {

        public function TextAreaDefaultProp()
        {
            super();
        }
        // Define a setter method to set the text property
        // to the value of the default property.
        public function set defaultText(value:String):void {
            if (value!=null)
            text=value;
        }
        public function get defaultText():String {
            return text;
        }
    }
}
```

In this example, you add a new property to the TextArea control, called `defaultText`, and specify it as the default property of the control. The setter method for `defaultText` just sets the value of the `text` property of the control. You can then use the default property in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainTextAreaDefaultProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <MyComp:TextAreaDefaultProp>Hello</MyComp:TextAreaDefaultProp>
</s:Application>
```

The one place where Flex prohibits the use of a default property is when you use the ActionScript class as the root tag of an MXML component. In this situation, you must use child tags to define the property, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/myComponents/TextAreaDefaultPropMXML.mxml -->
<MyComp:TextAreaDefaultProp xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
        <MyComp:defaultText>Hello</MyComp:defaultText>

</MyComp:TextAreaDefaultProp>
```

Making properties accessible in Flash Builder

You can make your property definitions accessible in Adobe Flex Builder by adding the `[Inspectable]` metadata tag to the property definition. For example, if you are using Flash Builder, you can insert the `[Inspectable]` metadata tag to define the property as user-editable (or *inspectable*), as the following example shows:

```
[Inspectable]
var prop1:Number;
```

You can also use the `[Inspectable]` metadata tag with setter and getter methods. For more information, see "Metadata tags in custom components" on page 2376.

Using data binding with custom properties

Data binding defines a syntax for automatically copying the value of a property of one object, the *source* property, to a property of another object, the *destination* property, at run time. Data binding is usually triggered when the value of the source property changes.

The following example shows a Text control that gets its data from a HSlider control's `value` property. The property name inside the curly braces ({ }) specifies a binding expression that copies the value of the source property, `mySlider.value`, into the destination property, the Text control's `text` property.

```
<mx:HSlider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

The current value of the HSlider control appears in the Text control when you stop moving the slider. To get continuous updates as you move the slider, set the `HSlider.liveDragging` property to `true`.

Using properties as the destination of a binding expression

Properties in your custom components can take advantage of data binding. Any property defined as a variable or defined by using a setter and getter method can automatically be used as the destination of a binding expression.

For example, in the section "Defining public properties in ActionScript" on page 2435, you created a class with the public property `maxFontSize`. You can use the `maxFontSize` property as the destination of a binding expression, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainTextAreaFontControlBindingDest.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <MyComp:TextAreaFontControl id="myTA"
        maxFontSize="{Number(myTI.text)}"/>
    <s:Label text="Enter max font size."/>
    <s:TextInput id="myTI" text="25"/>
</s:Application>
```

In this example, any value that the user enters into the TextInput control is automatically copied to the `maxFontSize` property.

Using properties as the source of a data binding expression

When a property is the source of a data binding expression, Flex automatically copies the value of the source property to any destination property when the source property changes. However, in order to signal to Flex to perform the copy, you must register the property with Flex and the source property must dispatch an event.

To register a property as a source for data bindings, you use the `[Bindable]` metadata tag. You can use this tag in three places:

- Before a class definition, in order to make all public properties defined as variables usable as the source of a binding expression. You can also do this by using setter and getter methods, in which case you use the `[Bindable]` tag before the getter method.
- Before a property that a variable defines, in order to make that specific property support binding
- Before a getter method for a property implemented by using setter and getter methods

Note: *When you use the* `[Bindable]` *metadata tag before a public class definition, it applies only to public properties; it does not apply to private or protected properties, or to properties defined in any other namespace. You must insert the* `[Bindable]` *metadata tag before a nonpublic property to make it usable as the source for a data binding expression.*

For more information on the `[Bindable]` metadata tag, see "Metadata tags in custom components" on page 2376.

The following example modifies the component in the section "Defining public properties in ActionScript" on page 2435 to make the `maxFontSize` and `minFontSize` properties usable as the source for data bindings:

```
    // Define public properties for tracking font size.
    [Bindable]
    public var maxFontSize:Number = 15;
    [Bindable]
    public var minFontSize:Number = 5;
```

If you omit the event name from the `[Bindable]` metadata tag, Flex automatically dispatches an event named `propertyChange` when the property changes to trigger the data binding. If the property value remains the same on a write operation, Flex does not dispatch the event or update the property.

When you define a property by using getter and setter methods so that the property is usable as the source for data binding, you include the `[Bindable]` metadata tag before the getter method, and optionally include the name of the event dispatched by the setter method when the property changes, as the following example shows:

```
package myComponents
{
    // createcomps_as/myComponents/TextAreaFontControlBinding.as
    import spark.components.TextArea;
    import flash.events.KeyboardEvent;
    import flash.events.Event;

    public class TextAreaFontControlBinding extends TextArea
    {
        public function TextAreaFontControlBinding()
        {
            super();
            addEventListener("keyDown", myKeyDown);
            addEventListener("creationComplete", myCreationComplete);
        }
        private var currentFontSize:Number;
        public var minFontSize:Number = 5;
        // Define private variable for maxFontSize.
        public var _maxFontSize:Number = 15;

        // Define public getter method, mark the property
        // as usable for the source of data binding,
        // and specify the name of the binding event.
        [Bindable("maxFontSizeChanged")]
        public function get maxFontSize():Number {
            return _maxFontSize;
        }
        // Define public setter method.
        public function set maxFontSize(value:Number):void {
            if (value <= 30) {
                _maxFontSize = value;
            } else _maxFontSize = 30;
            // Dispatch the event to trigger data binding.
            dispatchEvent(new Event("maxFontSizeChanged"));
        }

        private function myCreationComplete(eventObj:Event):void {
            // Get current font size
             currentFontSize = getStyle('fontSize');
        }
        // keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Was Ctrl key pressed?
            if (eventObj.ctrlKey)
            {
                switch (eventObj.keyCode) {
                    // Was Ctrl+Alt+I pressed?
                    case 73 :
                        if (currentFontSize < maxFontSize) {
                            currentFontSize = currentFontSize + 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    // Was Ctrl+Alt+M pressed?
                    case 77 :
                        if (currentFontSize > minFontSize) {
                            currentFontSize = currentFontSize - 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    default :
                        break;
                }
            }
        }

    }
}
```

In this example, the setter updates the value of the property, and then dispatches an event to trigger an update of any data binding destination. The name of the event is not restricted. You can use this component in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainTextAreaFontControlBindingSource.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <MyComp:TextAreaFontControlBinding id="myTA"
        maxFontSize="{Number(myTI.text)}"/>
    <s:Label text="Enter max font size."/>
    <s:TextInput id="myTI" text="15"/>

    <s:Label text="Current max font size."/>
    <s:TextArea text="{String(myTA.maxFontSize)}"/>
</s:Application>
```

Defining a method override

You can override a method of a base class in your ActionScript component. To override the method, you add a method with the same signature to your class, and prefix it with the `override` keyword. The following example overrides the `HGroup.addElement()` method to open an Alert box when a new item is added to it:

```
package myComponents
{
    // createcomps_as/myComponents/HGroupWithAlert.as
    import mx.controls.Alert;
    import spark.components.HGroup;
    import mx.core.IVisualElement;
    public class HGroupWithAlert extends HGroup
    {
        // Define the constructor.
        public function HGroupWithAlert() {
            super();
        }
      // Define the override.
      override public function addElement(child:IVisualElement):IVisualElement {

            // Call super.addChild().
            super.addElement(child);

            // Open the Alert box.
            Alert.show("Item added successfully");
            return child;
        }
    }
}
```

Notice that the method implementation calls the `super.addElement()` method. The call to `super.addElement()` causes Flex to invoke the superclass's `addElement()` method to perform the operation. Your new functionality to open the Alert box occurs after the `super.addElement()` method.

You might have to use `super()` to call the base class method before your code, after your code, or not at all. The location is determined by your requirements. To add functionality to the method, you call `super()` before your code. To replace the base class method, you do not call `super()` at all.

The following example uses this component in an application:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainHBoxWithAlert.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import spark.components.Button;
            public function addButton():void {
                var myButton:Button = new Button();
                myButton.label = "New Button";
                myHGroup.addElement(myButton);
            }
        ]]>
    </fx:Script>
    <MyComp:HGroupWithAlert id="myHGroup"/>
    <s:Button label="Add Button"
        click="addButton();"/>
</s:Application>
```

Initializing inherited properties with tag attributes in MXML

In an MXML component, you can initialize the value of any inherited public, writable property by defining a child tag of the MXML component with an `id` property that matches the name of the inherited property. For example, you define a custom Panel component based on the Flex Panel container, named MyPanel.as, as the following example shows:

```
package myComponents
{
    // createcomps_as/myComponents/MyPanel.as
    import spark.components.Panel;
    import spark.components.TextInput;
    public class MyPanel extends Panel {
        // Define public variables for two child components.
        public var myInput:TextInput;
        public var myOutput:TextInput;
        public function MyPanel() {
            super();
        }
        // Copy the text from one child component to another.
        public function xfer():void {
            myOutput.text = myInput.text;
        }
    }
}
```

In this example, the MyPanel component defines two variables corresponding to TextInput controls. You then create a custom MXML component, named MyPanelComponent.mxml, based on MyPanel.as, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_as/myComponents/MyPanelComponent.mxml -->
<MyComps:MyPanel xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComps="myComponents.*">
    <MyComps:layout>
        <s:VerticalLayout/>
    </MyComps:layout>
      <s:TextInput id="myInput"/>
      <s:TextInput id="myOutput"/>
</MyComps:MyPanel>
```

Notice that the value of the `id` property for the two TextInput controls matches the variable names of the properties defined in the MyPanel component. Therefore, Flex initializes the inherited properties with the TextInput controls that you defined in MXML. This technique for initializing properties can be referred to as *code behind*.

You can use your custom component in the following Flex application:

```
<?xml version="1.0"?>
<!-- createcomps_as/MainCodeBehindExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComps="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
   <MyComps:MyPanelComponent id="myP"/>
   <s:Button label="Copy" click="myP.xfer();"/>
</s:Application>
```

If the value of the `id` property of a TextInput control does not match an inherited property name, Flex creates a property of the component, where the `id` property defines the name of the new property.

To support initialization from MXML, an inherited property must have the following characteristics:

- The inherited property must be public.

  If you try to initialize a nonpublic inherited property, the Flex compiler issues an error.

- The inherited property must be writable.

  If you try to initialize a constant, or a property defined by a getter method without a corresponding setter method, the Flex compiler issues an error.

- The data type of the value that you specify to the inherited property must by compatible with the data type of the property.

  If you try to initialize a property with a value of an incompatible data type, the Flex compiler issues an error.

Defining events in ActionScript components

Flex components dispatch their own events and listen to other events. An object that wants to know about another object's events registers as a listener with that object. When an event occurs, the object dispatches the event to all registered listeners.

The core class of the Flex architecture, mx.core.UIComponent, defines core events, such as updateComplete, `resize`, `move`, `creationComplete`, and others that are fundamental to all components. Subclasses of these classes inherit and dispatch these events.

Custom components that extend existing Flex classes inherit all the events of the superclass. If you extend the Button class to create the MyButton class, you can use the events inherited from the Button class, such as `mouseOver` or `creationComplete`, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- createcomps_as/MainMyButtonAS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <fx:Script>
        <![CDATA[

            import flash.events.Event;

            private function handleClick(eventObj:Event):void {
                // Define event listener.
            }

            private function handleCreationComplete(eventObj:Event):void {
                // Define event listener.
            }

        ]]>
    </fx:Script>

    <MyComp:MyButton
        click="handleClick(event);"
        creationComplete="handleCreationComplete(event);"/>
</s:Application>
```

Your custom components can also define new events based on the requirements of your components. For example, the section “Using data binding with custom properties” on page 2440 showed how to define a custom event so that properties of your component can work with the Flex data binding mechanism.

Handling predefined events within the custom component

The previous example showed a custom component, MyButton, dispatching two events. In that example, you defined the event listeners in the main application file.

Your custom component can also define event listeners within the component itself to handle the events internally. For example, “Defining public properties as variables” on page 2436 defined event listeners for the `keyDown` and `creationComplete` events within the body of the component. This allows the component to handle those events internally.

***Note:** Even though you define event listeners for the events in the component itself, your application can also register listeners for those events. The event listeners defined within the component execute before any listeners defined in the application.*

The example used the `creationComplete` event to access the default `fontSize` property of the component. You could not access this property in the constructor itself because Flex does not define it until after the component is created. For more information on the initialization order of a component, see "Create advanced MX visual components in ActionScript" on page 2475.

Dispatching custom events

Your ActionScript component can define custom events and use the predefined events. You use custom events to support data binding, to respond to user interactions, or to trigger actions by your component. For an example that uses events to support data binding, see "Using data binding with custom properties" on page 2440.

For each custom event dispatched by your component, you must do the following:

1. Create an Event object describing the event.
2. (Optional) Use the `[Event]` metadata tag to make the event public so that other components can listen for it.
3. Dispatch the event by using the dispatchEvent() method.

To add information to the event object, you define a subclass of the flash.events.Event class to represent the event object. For more information on creating custom event classes, see "Custom events" on page 2369.

You might define some custom events that are used internally by your component, and are not intended to be recognized by the other components. For example, the following component defines a custom event, dispatches it, and handles it all within the component:

```
package myComponents
{
    // createcomps_as/myComponents/ModalTextEvent.as
    import spark.components.TextArea;
    import flash.events.Event;
    [Event(name="enableChanged", type="flash.events.Event")]
    public class ModalTextEvent extends TextArea {
        public function ModalTextEvent() {
            super();

            // Register event listener.
            addEventListener("enableChanged", enableChangedListener);
        }
        public function enableInput(value:Boolean):void {
            // Method body.

            // Dispatch event.
            dispatchEvent(new Event("enableChanged"));
        }
        private function enableChangedListener(eventObj:Event):void {
            // Handle event.
        }
    }
}
```

In this example, the public method `enableInput()` lets the user enable or disable input to the control. When you call the `enableInput()` method, the component uses the `dispatchEvent()` method to dispatch the `enableChanged` event. The `dispatchEvent()` method has the following signature:

```
dispatchEvent(eventObj)
```

The *eventObj* argument is the event object that describes the event.

If you want an MXML component to be able to register a listener for the event, you must make the event known to the Flex compiler by using the [Event] metadata tag. For each public event that your custom component dispatches, you add an [Event] metadata keyword before the class definition that defines that event, as the following example shows:

```
[Event(name="enableChanged", type="flash.events.Event")]
public class ModalTextEvent extends TextArea {
    ...
}
```

If you do not identify an event in the class file with the [Event] metadata tag, the compiler generates an error when an MXML component attempts to register a listener for that event. Any component can register an event listener for the event in ActionScript using the addEventListener() method, even if you omit the [Event] metadata tag.

You can then handle the event in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/ASMainModalTextEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComps="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            private function handleEnableChanged(event:Event):void {
                myTA.text="Got Event";
            }
        ]]>
    </fx:Script>
    <MyComps:ModalTextEvent id="myMT"
        enableChanged="handleEnableChanged(event);"/>

    <s:Button click="myMT.enableInput(true);"/>
    <s:TextArea id="myTA"/>
</s:Application>
```

Applying styles to custom components

Style properties define the look of a component, from the size of the fonts used to the color of the background. Your custom ActionScript components inherit all of the styles of the base class, so you can set them in the same way as for that base class.

To change style properties in custom components, use the setStyle() method in the component's constructor. This applies the same style to all instances of the component, but users of the component can override the settings of the setStyle() method in MXML tags. Any style properties that are not set in the component's class file are inherited from the component's superclass.

The following ActionScript class file sets the color and fontWeight styles of the BlueButton control:

```
package myComponents
{
    // createcomps_as/myComponents/BlueButton.as
    import spark.components.Button;
    public class BlueButton extends Button
    {

        public function BlueButton() {
            super();
            // Set the label text to blue.
            setStyle("color", 0x0000FF);
            // Set the borderColor to blue.
            setStyle("fontWeight", "bold");
        }
    }
}
```

The following MXML file uses the BlueButton control with the default `color` and `fontWeight` styles set in your component's class file:

```
<?xml version="1.0"?>
<!-- createcomps_as/ASMainBlueButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComps="myComponents.*">
    <MyComps:BlueButton label="Submit"/>

</s:Application>
```

Setting the styles in a constructor does not prevent users of the component from changing the style. For example, the user could still set their own value for the `color` style, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/MainBlueButtonRed.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComps="myComponents.*">
    <MyComps:BlueButton label="Submit" color="0xFF0000"/>

</s:Application>
```

In addition to setting the `color` property, you can set the font face, font size, and other style properties. For more information on the available style properties, see the parent control's class information.

You can also define new style properties for your components. For more information, see "Custom style properties" on page 2501.

Applying styles from a defaults.css file

If you package your component in a SWC file, you can define a global style sheet, named defaults.css, in the SWC file. The defaults.css file defines the default style settings for all of the components defined in the SWC file.

For more information, see "Applying styles from a defaults.css file" on page 2409.

Create advanced Spark visual components in ActionScript

You can create advanced visual components for use in applications built in Adobe® Flex®. Advanced visual components override methods of the UIComponent or SkinnableComponent base classes.

About creating advanced Spark components

Simple visual components are subclasses of existing Flex components that modify the appearance of the component by using skins or styles, or add new functionality to the component. For example, you add a new event type to a Button control, or modify the default styles or skins of a DataGrid control. For more information, see "Create simple visual components in ActionScript" on page 2433.

In advanced components, you typically perform the following actions:

- Modify the basic functionality or logic of an existing component.
- Create a composite component that encapsulates two or more components within it.
- Create a skinnable component by creating a subclass of the SkinnableComponent class.

  A skinnable component uses two files: one for the component definition and one for the skin definition. Create a Spark skinnable component as a subclass of the SkinnableComponent class.
- Create a nonskinnable component by creating a subclass of the UIComponent or other nonskinnable Spark class.

  A nonskinnable component is defined by a single file. Create a nonskinnable component as a subclass of the UIComponent class. You can also create one from a Spark class that does not have the SkinnableComponent class in its class hierarchy, such a Group or DataGroup.

This topic contains several examples of Spark ActionScript components. For more examples, examine the source code for the Spark components in your Flex installation directory. The spark.components and spark.components.supportClasses packages contain many of the Spark components mentioned in this topic.

Spark skinnable components and skin classes

When creating an skinnable Spark component in ActionScript, you create two classes: the component class and the skin class.

The component class defines the core behavior of the component. This behavior includes defining the events dispatched by the component, the data that the component represents, the skin parts implemented by the skin class, and the view states that the skin class supports.

The skin class manages the visual appearance of the component and creates visual subcomponents. The skin class defines the default layout of the component, its default size, the supported view states, graphics, and data representation.

About overriding protected UIComponent methods for Spark components

All Flex visual components are subclasses of the UIComponent class. Therefore, visual components inherit the methods, properties, events, styles, and effects defined by the UIComponent class.

To create an advanced visual component, you must implement a class constructor. Also, you optionally override one or more of the following protected methods of the UIComponent class:

UIComponent method	Description
`commitProperties()`	Commits any changes to component properties, either to make the changes occur at the same time or to ensure that properties are set in a specific order. For more information, see "Implementing the commitProperties() method for Spark components" on page 2458.
`createChildren()`	Creates any child components of the component. For example, the Halo ComboBox control contains a Halo TextInput control and a Halo Button control as child components. Typically, you do not implement this method in a Spark component because any child components are defined in the skin class. This topic does not describe how to implement the `createChildren()` method. For more information, see the example for creating a Halo component in "Implementing the createChildren() method for MX components" on page 2482.
`measure()`	Sets the default size and default minimum size of the component. You typically do not have to implement this method for Spark components. The default size of a Spark component is defined by the skin class, and by the children of the skin class. You also set the minimum and maximum sizes of the component in the root tag of the skin class. This topic does not describe how to implement the `measure()` method. For more information, see the example for creating a MX components in "Implementing the measure() method for MX components" on page 2485.
`updateDisplayList()`	Sizes and positions the children of the component on the screen based on all previous property and style settings, and draws any skins or graphic elements used by the component. The parent container for the component determines the size of the component itself. Typically, you do not have to implement this method for Spark components. A few Spark components, such as spark.components.supportClasses.SkinnableComponent, do implement it. The SkinnableComponent class implements it to pass sizing information to the component's skin class. This topic does not describe how to implement the `updateDisplayList()` method. For more information, see the example for creating a Halo component in "Implementing the updateDisplayList() method for MX components" on page 2489.

Component users do not call these methods directly; Flex calls them as part of the initialization process of creating a component, or when other method calls occur. For more information, see "About the component instantiation life cycle for MX components" on page 2477.

About overriding SkinnableComponent methods for Spark components

All Spark skinnable components are subclasses of the SkinnableComponent class. Therefore, skinnable components inherit the methods, properties, events, styles, and effects defined by the SkinnableComponent class.

To create an skinnable Spark component, you optionally override one or more of the following methods of the SkinnableComponent class:

SkinnableComponent method	Description

`attachSkin()`
`detachSkin()`
	Called automatically by the `commitProperties()` method when a skin is added, `attachSkin()`, or a skin is removed, `detachSkin()`. You can optionally implement these methods to add a specific behavior to a skin.
	Typically you do not implement these methods.

`partAdded()`
`partRemoved()`
	Called automatically when a skin part is added or removed. You typically override `partAdded()` to attach event handlers to the skin part, configure the skin part, or perform other actions when a skin part is added. Implement the `partRemoved()` method to remove the even handlers added in `partAdded()`.
	For more information, see "Implementing the partAdded() and partRemoved() methods for skinnable components for Spark components" on page 2462.

`getCurrentSkinState()`	Called automatically by the `commitProperties()` method to set the view state of the skin class. For more information, see "Implementing the getCurrentSkinState() method for skinnable components for Spark components" on page 2463.

Component users do not call these methods directly; Flex calls them as part of the initialization process of creating a component, or when other method calls occur. For more information, see "About the component instantiation life cycle for MX components" on page 2477.

About the invalidation methods for Spark components

During the lifetime of a component, your application might modify the component by changing its size or position, modifying a property that controls its display, or modifying a style or skin property of the component. For example, you might change the font size of the text displayed in a component. As part of changing the font size, the component's size might also change, which requires Flex to update the layout of the application. The layout operation might require Flex to invoke the `commitProperties()`, `measure()`, `getCurrentSkinState()`, and the `updateDisplayList()` methods of your component.

Your application can programmatically change the font size of a component much faster than Flex can update the layout of an application. Therefore, you only want to update the layout after you are sure that you've determined the final value of the font size.

In another scenario, when you set multiple properties of a component, such as the `label` and `icon` properties of a Button control, you want the `commitProperties()`, `measure()`, `getCurrentSkinState()`, and `updateDisplayList()` methods to execute only once, after all properties are set. You do not want these methods to execute when you set the `label` property, and then execute again when you set the `icon` property.

Also, several components might change their font size at the same time. Rather than updating the application layout after each component changes its font size, you want Flex to coordinate the layout operation to eliminate any redundant processing.

Flex uses an invalidation mechanism to synchronize modifications to components. Flex implements the invalidation mechanism as a set of methods that you call to signal that something about the component has changed and requires Flex to call the component's `commitProperties()`, `measure()`, `getCurrentSkinState()`, or `updateDisplayList()` methods.

The following table describes the invalidation methods:

Invalidation method	**Description**
`invalidateDisplayList()`	Marks a component so that its `updateDisplayList()` method gets called during the next screen update.
`invalidateProperties()`	Marks a component so that its `commitProperties()` method gets called during the next screen update.
`invalidateSize()`	Marks a component so that its `measure()` method gets called during the next screen update.
`invalidateSkinState()`	Marks a component so that its `commitProperties()` method gets called on the next screen update to change the view state of the skin class. The `commitProperties()` method calls the `getCurrentSkinState()` method.

When a component calls an invalidation method, it signals to Flex that the component must be updated. When multiple components call invalidation methods, Flex coordinates updates so that they all occur together during the next screen update.

Typically, component users do not call the invalidation methods directly. Instead, they are called by the component's setter methods, or by any other methods of a component class as necessary. For more information and examples, see "Implementing the commitProperties() method for MX components" on page 2483.

About the Spark component instantiation life cycle

The component instantiation life cycle describes the sequence of steps that occur when you create a component object from a component class. As part of that life cycle, Flex automatically calls component methods, dispatches events, and makes the component visible.

The following example creates a Button control in ActionScript and adds it to a Group container:

```
// Create a Group container.
var groupContainer:Group = new Group();
// Configure the Group container.
groupContainer.x = 10;
groupContainer.y = 10;

// Create a Button control.
var b:Button = new Button()
// Configure the button control.
b.label = "Submit";
...
// Add the Button control to the Box container.
groupContainer.addElement(b);
```

The following steps show what occurs when you execute the code to create the Button control, and add the control to the container:

1 You call the component's constructor, as the following code shows:

```
// Create a Button control.
var b:Button = new Button()
```

2 You configure the component by setting its properties, as the following code shows:

```
// Configure the button control.
b.label = "Submit";
```

Component setter methods might call the `invalidateProperties()`, `invalidateSize()`, `invalidateSkinState()`, or `invalidateDisplayList()` methods.

3 You call the `addElement()` method to add the component to its parent, as the following code shows:

```
// Add the Button control to the Box container.
gropContainer.addElement(b);
```

Flex then performs the following actions:

4 Sets the `parent` property for the component to reference its parent container.

5 Computes the style settings for the component.

6 Dispatches the `preinitialize` event on the component.

7 Calls the component's `createChildren()` method. For skinnable components, this causes a call to `attachSkin()`, which calls `partAdded()`, for all static parts defined in the skin file.

8 Calls the `invalidateProperties()`, `invalidateSize()`, `invalidateSkinSate()`, and `invalidateDisplayList()` methods to trigger calls to the `commitProperties()`, `measure()`, `getCurrentSkinState()`, or `updateDisplayList()` methods during the next `render` event.

The only exception to this rule is that Flex does not call the `measure()` method when the user sets the height and width of the component.

9 Dispatches the `initialize` event on the component. At this time, all of the component's children are initialized, but the component has not been sized or processed for layout. You can use this event to perform additional processing of the component before it is laid out.

Because the `initialize` event is dispatched early in the component's startup sequence, make sure that none of your processing causes the component to invalidate itself. You typically perform any final processing during the `creationComplete` event.

10 Dispatches the `elementAdd` event on the parent container.

11 Dispatches the `initialize` event on the parent container.

12 During the next `render` event, Flex performs the following actions:

a Calls the component's `commitProperties()` method. For skinnable components, the `commitProperties()` method calls the `getCurrentSkinState()` methods.

b Calls the component's `measure()` method.

c Calls the component's `updateDisplayList()` method.

13 Flex dispatches additional `render` events if the `commitProperties()`, `measure()`, or `updateDisplayList()` methods call the `invalidateProperties()`, `invalidateSize()`, `invalidateSkinSate()`, or `invalidateDisplayList()` methods.

14 After the last `render` event occurs, Flex performs the following actions:

a Makes the component visible by setting the `visible` property to `true`.

b Dispatches the `creationComplete` event on the component. The component is sized and processed for layout. This event is only dispatched once when the component is created.

c Dispatches the `updateComplete` event on the component. Flex dispatches additional `updateComplete` events whenever the layout, position, size, or other visual characteristic of the component changes and the component is updated for display.

Most of the work for configuring a component occurs when you add the component to a container by using the `addElement()` method. That is because until you add the component to a container, Flex cannot determine its size, set inheriting style properties, or draw it on the screen.

You can also define your application in MXML, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:Button label="Submit"/>
    </s:Group>
</s:Application>
```

The sequence of steps that Flex executes when creating a component in MXML are equivalent to the steps described for ActionScript.

You can remove a component from a container by using the `removeElement()` method. If there are no references to the component, it is eventually deleted from memory by the garbage collection mechanism of Adobe® Flash® Player or Adobe® AIR™.

About the steps for creating a Spark component

When you implement a component, you override component methods, define new properties, dispatch new events, or perform any other customizations required by your application.

To implement your component, follow these general steps:

1 Create the skin class for the component. You typically create the skin class in MXML. For more information on skins, see "Spark Skinning" on page 1602.

2 Create the component's ActionScript class file.

   a For a skinnable component, extend one of the base classes, such as SkinnableComponent, or a subclass of class SkinnableComponent. For a nonskinnable component, extend UIComponent or a Spark class that does not have SkinnableComponent in it class hierarchy.

   b Implement the constructor.

   c Implement the UIComponent.createChildren() method. You rarely have to implement this method for Spark components.

   d Implement the UIComponent.commitProperties() method.

   e Implement the UIComponent.measure() method. You rarely have to implement this method for Spark components.

   f Implement the UIComponent.updateDisplayList() method. You rarely have to implement this method for Spark components.

   g For a skinnable component, implement the SkinnableComponent.partAdded() and SkinnableComponent.partRemoved() methods.

   h For a skinnable component, implement the SkinnableComponent.getCurrentSkinState() method.

   i Add properties, methods, styles, events, and metadata.

3 Deploy the component as an ActionScript file or as a SWC file.

For more information about MXML tag properties and embedding graphic and skin files, see "Create simple visual components in ActionScript" on page 2433.

You do not have to override all component methods to define a new component. You only override the methods required to implement the functionality of your component. If you create a subclass of an existing component, such as Button, you implement only the methods necessary for you to add any new functionality to the component.

About interfaces

Flex uses interfaces to divide the basic functionality of components into discrete elements so that they can be implemented piece by piece. For example, to make your component focusable, it must implement the IFocusManagerComponent interface; to let it participate in the layout process, it must implement ILayoutClient interface.

To simplify the use of interfaces, the UIComponent class implements all of the interfaces defined in the following table, except for the IFocusManagerComponent interface. However, many subclasses of UIComponent implement the IFocusManagerComponent interface.

Therefore, if you create a subclass of the UIComponent class or subclass of UIComponent, such as SkinnableComponent, you do not have to implement these interfaces. But, if you create a component that is not a subclass of UIComponent, and you want to use that component in Flex, you might have to implement one or more of these interfaces.

The following table lists the main interfaces implemented by Flex components:

Interface	Use
IAdvancedStyleClient	Indicates that the component supports the advanced style subsystem.
IAutomationObject	Indicates that a component is an object within the automation object hierarchy.
IChildList	Indicates the number of children in a container.
IConstraintClient	Indicates that the component support layout constraints.
IDeferredInstantiationUIComponent	Indicates that a component or object can effect deferred instantiation.
IFlexDisplayObject	Specifies the interface for skin elements.
IFlexModule	indicates that the component can be used with module factories
IFocusManagerComponent	Indicates that a component or object is focusable, which means that the components can receive focus from the FocusManager. The UIComponent class does not implement IFocusable because some components are not intended to receive focus.
IInvalidating	Indicates that a component or object can use the invalidation mechanism to perform delayed, rather than immediate, property commitment, measurement, and drawing or layout.
ILayoutManagerClient	Indicates that a component or object can participate in the LayoutManager's commit, measure, and update sequence.
IPropertyChangeNotifier	Indicates that a component supports a specialized form of event propagation.
IStateClient	Indicates that the component supports view states.
IToolTipManagerClient	Indicates that a component has a `toolTip` property, and therefore is monitored by the ToolTipManager.
IUIComponent	Defines the basic set of APIs that you must implement in order to be a child of layout containers and lists.
IValidatorListener	Indicates that a component can listen for validation events, and therefore show a validation state, such as a red border and error tooltips.
IVisualElement	Indicates that the component can be laid out and displayed in a Spark application.

Implementing a Spark component

When you create a custom component in ActionScript, you have to override the methods of UIComponent and, if the component is skinnable, of SkinnableComponent.

Basic component structure

The following example shows the basic structure of a skinnable Spark component:

```
package myComponents
{
    public class MyComponent extends SkinnableComponent
    {
        ....
    }
}
```

You must define your ActionScript custom components within a package. The package reflects the directory location of your component within the directory structure of your application.

The class definition of your component must be prefixed by the `public` keyword. A file that contains a class definition can have one, and only one, public class definition, although it can have additional internal class definitions. Place any internal class definitions at the bottom of your source file below the closing curly brace of the package definition.

Implementing the constructor

Your ActionScript class should define a public constructor method for a class that is a subclass of the UIComponent class, or a subclass of any child of the UIComponent class. The constructor has the following characteristics:

- No return type
- Declared public
- No arguments
- Calls the `super()` method to invoke the superclass' constructor

Each class can contain only one constructor method; ActionScript does not support overloaded constructor methods. For more information, see "Defining the constructor" on page 2366.

Use the constructor to set the initial values of class properties. For example, you can set default values for properties and styles, or initialize data structures, such as Arrays. You can also set the `skinClass` style to the name of your skin class.

Do not create child display objects in the constructor; you should use it only for setting initial properties of the component. If your component creates child components, create them in the skin class.

Implementing the commitProperties() method for Spark components

You use the commitProperties() method to coordinate modifications to component properties. Most often, you use it with properties that affect how a component appears on the screen.

Flex schedules a call to the `commitProperties()` method when a call to the `invalidateProperties()` method occurs. The `commitProperties()` method executes during the next `render` event after a call to the `invalidateProperties()` method. When you use the `addElement()` method to add a component to a container, Flex automatically calls the `invalidateProperties()` method.

The typical pattern for defining component properties is to define the properties by using getter and setter methods, as the following example shows:

```
// Define a private variable for the alignText property.
private var _alignText:String = "right";

// Define a flag to indicate when the _alignText property changes.
private var bAlignTextChanged:Boolean = false;

// Define getter and setter methods for the property.
public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;

    // Trigger the commitProperties(), measure() method.
    invalidateProperties();
    invalidateSize();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // Handle alignment change
        ...

        // If necessary, call invalidateDisplayList() to update the display.
        invalidateDisplayList();
    }
}
```

As you can see in this example, the setter method modifies the property, calls the `invalidateProperties()` method, and then returns. The setter itself does not perform any calculations based on the new property value. This design lets the setter method return quickly, and leaves any processing of the new value to the `commitProperties()` method.

The `commitProperties()` method in the previous example process the changes to the property, then calls the `invalidateDisplay()` method to cause the component to update its display. The call to the `invalidateDisplay()` method is only necessary if the component has to update its display based on the property change.

The main advantages of using the `commitProperties()` method are the following:

- To coordinate the modifications of multiple properties so that the modifications occur synchronously.

  For example, you might define multiple properties that control the text displayed by the component, such as the alignment of the text within the component. A change to either the text or the alignment property requires Flex to update the appearance of the component. However, if you modify both the text and the alignment, you want Flex to perform any calculations for sizing or positioning the component once, when the screen updates.

  Therefore, you use the `commitProperties()` method to calculate any values based on the relationship of multiple component properties. By coordinating the property changes in the `commitProperties()` method, you can reduce unnecessary processing overhead.

- To coordinate the sequence that you set properties.

  Some properties may have to be set in a particular sequence. In the `commitProperties()` method, determine which properties are being modified, and, if necessary, set them in the proper order.

- To coordinate multiple modifications to the same property.

  You do not necessarily want to perform a complex calculation every time a user updates a component property. For example, users modify the `icon` property of the Button control to change the image displayed in the button. Calculating the label position based on the presence or size of an icon can be a computationally expensive operation that you want to perform only when necessary.

  To avoid this behavior, you use the `commitProperties()` method to perform the calculations. Flex calls the `commitProperties()` method when it updates the display. That means you perform the calculations once when Flex updates the screen, regardless of the number of times the property changed between screen updates.

The following example shows how you can handle two related properties in the `commitProperties()` method:

```
// Define a private variable for the text property.
private var _text:String = "ModalText";
private var bTextChanged:Boolean = false;

// Define the getter method.
public function get text():String {
        return _text;
}

//Define the setter method to call invalidateProperties()
// when the property changes.
public function set text(t:String):void {
    _text = t;
    bTextChanged = true;
    invalidateProperties();
}

// Define a private variable for the alignText property.
private var _alignText:String = "right";
private var bAlignTextChanged:Boolean = false;

public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;
    invalidateProperties();
    invalidateSize();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flags indicate a change to both properties.
    if (bTextChanged && bAlignTextChanged) {
        // Reset flags.
        bTextChanged = false;
        bAlignTextChanged = false;

        // If necessary, update the dispaly.
        invalidateDisplayList();
    }

    // Check whether the flag indicates a change to the text property.
    if (bTextChanged) {
        // Reset flag.
        bTextChanged = false;

        // If necessary, update the dispaly.
        invalidateDisplayList();
    }

    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // If necessary, update the dispaly.
        invalidateDisplayList();
    }
}
```

Implementing the updateDisplayList() method for Spark components

The updateDisplayList() method sizes and positions parts of the component based on all previous property and style settings. The parent container for the component determines the size of the component itself. You rarely have to implement this method for Spark components.

A component does not appear on the screen until its `updateDisplayList()` method gets called. Flex schedules a call to the `updateDisplayList()` method when a call to the `invalidateDisplayList()` method occurs. The `updateDisplayList()` method executes during the next `render` event after a call to the `invalidateDisplayList()` method. When you use the `addElement()` method to add a component to a container, Flex automatically calls the `invalidateDisplayList()` method.

In general, all visual aspects of a Spark component are controlled by the skin class. But, there are times when the component must participate in the visual display. The only time a Spark component implements the `updateDisplayList()` method is when the component has view-specific knowledge that it must control. For example, the slider thumb of the Spark HSlider and VSlider controls is positioned by the `updateDisplayList()` method.

Make sure to perform as much visual display as possible in the skin. For example, the `updateDisplayList()` method of the HSlider component only sets the x position of the slider thumb. The skin class sets the y position.

The main uses of the `updateDisplayList()` method are the following:

- To set the size and position of the elements of the component for display.

  Many components are made up of one or more child components, or have properties that control the display of information in the component.

  To size components in the `updateDisplayList()` method, you use the `setActualSize()` method, not the sizing properties, such as `width` and `height`. To position a component, use the `move()` method, not the `x` and `y` properties.

- To draw any visual elements necessary for the component.

Components support many types of visual elements such as graphics, styles, and borders. Within the `updateDisplayList()` method, you can add these visual elements, use the Flash drawing APIs, and perform additional control over the visual display of your component.

The `updateDisplayList()` method has the following signature:

```
protected function updateDisplayList(unscaledWidth:Number,
    unscaledHeight:Number):void
```

The properties have the following values:

unscaledWidth Specifies the width of the component, in pixels, in the component's coordinates, regardless of the value of the `scaleX` property of the component. This is the width of the component as determined by its parent container.

unscaledHeight Specifies the height of the component, in pixels, in the component's coordinates, regardless of the value of the `scaleY` property of the component. This is the height of the component as determined by its parent container.

Scaling occurs in Flash Player or AIR, after `updateDisplayList()` executes. For example, a component with an `unscaledHeight` value of 100, and with a `scaleY` property of 2.0, appears 200 pixels high in Flash Player or AIR.

Implementing the partAdded() and partRemoved() methods for skinnable components for Spark components

Some skinnable components are composed of one or more subcomponents. For example, a NumericStepper component contains a subcomponent for an up button, a down button, and a text area.

The component class is responsible for controlling the behavior of the subcomponents. The skin class is responsible for defining the subcomponents, including the appearance of the component, its subcomponents, and any other visual aspects of the component.

Flex clearly defines the relationship between the component class and the skin class. The component class must do the following:

- Identify the skin parts that it expects with the `[SkinPart]` metadata tag. The skin parts are implemented in the skin file. For more information on using the `[SkinPart]` metadata tag, see "SkinPart metadata tag" on page 2389.
- Identify the view states that the component supports with the `[SkinStates]` metadata tag. For more information on the `[SkinState]` metadata tag, see "SkinState metadata tag" on page 2390.
- Use CSS styles to associate the skin class with the component.

The skin class must do the following:

- Specify the component name with the `[HostComponent]` metadata tag. While the `[HostComponent]` metadata tag is not required, it is strongly recommended. For more information on the `[HostComponent]` metadata tag, see "HostComponent metadata tag" on page 2385.
- Declare the view states, and define their appearance.
- Define the appearance of the skin parts. The skin parts must use the same name as the corresponding skin-part property in the component.

Flex calls the `partAdded()` and `partRemoved()` methods automatically when a skin is created or destroyed. You typically override the `partAdded()` method to attach event handlers to a skin part, configure a skin part, or perform other actions when a skin part is added. You implement the `partRemoved()` method to remove the even handlers added in `partAdded()`.

In the component class, define skin parts as properties. In the following example, the component defines two required skin parts:

```
// Define the skin parts.
[SkinPart(required="true")]
public var modeButton:Button;

[SkinPart(required="true")]
public var textInput:RichEditableText;
```

The first skin part defines a Button control, and the second defines a RichEditableText control. While the skin parts are defined as properties of the component, component users do not directly modify them. The skin class defines their implementation and appearance. For more information on defining skin parts, see "Skin parts" on page 1615.

In your implementation of the `partAdded()` method, you first use `super` to invoke the `partAdded()` method of the base class. Then, determine the skin part that was added, and configure it. Use the property name of the skin part to reference it. In this example, you set properties on the skin part and add event listeners to it:

```
override protected function partAdded(partName:String, instance:Object):void {
    super.partAdded(partName, instance);

    if (instance == textInput) {
        textInput.editable = false;
        textInput.text= _text;
        textInput.addEventListener("change", modeButton_clickHandler);
    }

    if (instance == modeButton) {
        modeButton.label = "Toggle Editing Mode";
        modeButton.addEventListener("click", modeButton_changeHandler);
    }
}
```

In your implementation of the `partRemoved()` method, use `super` to invoke the `partRemoved()` method of the base class. You can then remove the event listeners added by the `partAdded()` method:

```
override protected function partRemoved(partName:String, instance:Object):void {
    super.partRemoved(partName, instance);

    if (instance == textInput) {
        textInput.removeEventListener("change", modeButton_clickHandler);
    }

    if (instance == modeButton) {
        modeButton.removeEventListener("click", modeButton_changeHandler);
    }
}
```

Implementing the getCurrentSkinState() method for skinnable components for Spark components

A component must identify the view states that its skin supports. Use the `[SkinState]` metadata tag to define the view states in the component class. This tag has the following syntax:

```
[SkinState("stateName")]
```

Specify the metadata before the class definition. For more information on the `[SkinState]` metadata tag, see "SkinState metadata tag" on page 2390.

The following example defines two view states for the component named ModalTextStates:

```
[SkinState("normal")]
[SkinState("textLeft")]
public class ModalTextStates extends SkinnableComponent
{
    ...
```

The component's skin class then define these view states, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    minWidth="100" minHeight="25">

    <s:states>
        <s:State name="normal" />
        <s:State name="textLeft"/>
    </s:states>
    ...
```

In your component, implement the `getCurrentSkinState()` method to set the view state of the skin class. Flex calls the `getCurrentSkinState()` method automatically from the `commitProperties()` method.

The `getCurrentSkinState()` method takes no arguments, and returns a String identifying the view state of the skin. You can use information in the class to determine the new view state. In the following example, you examine the `_textPlacement` property of the component to determine the view state of the skin:

```
override protected function getCurrentSkinState():String {
    var returnState:String = "normal";

    // Use information in the class to determine the new view state of the skin class.
    if (_textPlacement == "left") {
        returnState = "textLeft";
    }
    return returnState;
}
```

Adding version numbers

When releasing components, you can define a version number. This lets developers know whether they should upgrade, and helps with technical support issues. When you set a component's version number, use the static variable `version`, as the following example shows:

```
static var version:String = "1.0.0.42";
```

***Note:** Flex does not use or interpret the value of the* `version` *property.*

If you create many components as part of a component package, you can include the version number in an external file. That way, you update the version number in only one place. For example, the following code imports the contents of an external file that stores the version number in one place:

```
include "../myPackage/ComponentVersion.as"
```

The contents of the ComponentVersion.as file are identical to the previous variable declaration, as the following example shows:

```
static var version:String = "1.0.0.42";
```

Best practices when designing a component

Use the following practices when you design a component:

- Keep the file size as small as possible.
- Make your component as reusable as possible by generalizing functionality.
- Assume an initial state. Because style properties are on the object, you can set initial settings for styles and properties so your initialization code does not have to set them when the object is constructed, unless the user overrides the default state.

Example: Creating a skinnable Spark component

Composite components are components that contain multiple subcomponents. They might be graphical assets or a combination of graphical assets and component classes. For example, you can create a component that includes as subcomponents a button, rich text field, and a border graphic. Or, you can create a component that includes a text field and a validator.

When you create composite components, you define the subcomponents inside the component's skin class. You must plan the layout of the subcomponents that you are including, and set properties such as default values in the skin class.

Properties of the individual subcomponents are not directly accessible in MXML. For example, if you create a component that extends the SkinnableComponent class and uses a Button and a RichEditableText component as subcomponents, you cannot set the Button control's `label` property in MXML.

Instead, you can define a myLabel property on the custom component that is exposed in MXML. When the user sets the myLabel property, your custom component can set that property on the Button.

Creating the ModalText component

This example component, called ModalText and defined in the file ModalText.as, combines a Button control and a RichEditableText component. You use the Button control to enable or disable text input in the RichEditableText component.

This control has the following attributes:

- By default, you cannot edit the RichEditableText component.
- Click the Button control to toggle editing of the RichEditableText component.
- Use the `ModalText.text` property to programmatically write content to the RichEditableText component.
- Editing the text in the RichEditableText component dispatches the `change` event.
- Use the `text` property as the source for a data binding expression.

The following is an example MXML file that uses the ModalText control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark/SparkMainModalText.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <MyComp:ModalText/>

</s:Application>
```

Component users cannot directly access the properties of the Button and RichEditableText subcomponents. However, they can use descendant selectors to set the styles on the subcomponents, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark/SparkMainModalTextStyled.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <fx:Style>
        @namespace MyComp "myComponents.*";
        @namespace s "library://ns.adobe.com/flex/spark";
        MyComp|ModalText s|Button {
            chromeColor: #663366;
            color: #9999CC;
        }
    </fx:Style>

    <MyComp:ModalText/>

</s:Application>
```

In this example, you use descendent selectors to set the set the `chromeColor` and `color` styles of the Button subcomponent. For more information, see “Using Cascading Style Sheets” on page 1504.

Defining event listeners for composite components

Subcomponents can dispatch events. The main component can either handle the events internally, or propagate the event so that a component user can handle them.

Custom components implement the `partAdded()` method to create children of the component, as the following example shows:

```
override protected function partAdded(partName:String, instance:Object):void {
    super.partAdded(partName, instance);
    if (instance == textInput) {
        textInput.editable = false;
        textInput.text= _text;
        textInput.addEventListener("change", textInput_changeHandler);
    }
    if (instance == modeButton) {
        modeButton.label = "Toggle Editing Mode";
        modeButton.addEventListener("click", modeButton_changeHandler);
    }
}
```

The `partAdded()` method contains a call to the `addEventListener()` method to register an event listener for the `change` event generated by the RichEditableText subcomponent, and for the `click` event for the Button subcomponent. These event listeners are defined within the ModalText class, as the following example shows:

```
// Handle events for a change to RichEditableText.text property.
private function textInput_changeHandler(eventObj:Event):void {
        dispatchEvent(new Event("change"));
}

// Handle the click event for the Button subcomponent.
private function modeButton_changeHandler(eventObj:Event):void {
        text_mc.editable = !text_mc.editable;
}
```

You can handle an event dispatched by a child of a composite component in the component. In this example, the event listener for the Button subcomponent's `click` event is defined in the class definition to toggle the `editable` property of the RichEditableText subcomponent.

However, if a child component dispatches an event, and you want that opportunity to handle the event outside of the component, you must add logic to your custom component to propagate the event. Notice that the event listener for the `change` event for the RichEditableText subcomponent propagates the event. This lets you handle the event in your application, as the following example shows:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*">

    <fx:Script>
        <![CDATA[

            import flash.events.Event;

            function handleText(eventObj:Event)
            {
                ...
            }
        ]]>
    </fx:Script>

    <MyComp:ModalText change="handleText(event);"/>
</s:Application>
```

Creating the skin class for the ModalText component

The ModalText component defines a Button subcomponent and a Rich TextArea subcomponent. Each of these subcomponents is defined in the ModalText.as file as a required skin part.

A component uses CSS styles to specify the skin class that implements the skin parts. The ModalText component uses the `setStyle()` method to set the style. However, calling the `setStyle()` method in the component definition does not make it easy to reskin the component. You can also use an external style sheet or other mechanism to set the style that defines the skin class.

The skin class performs the following:

- Uses the `[HostComponent]` metadata tag to specify ModalText as the host component of the skin.
- Defines a single view state named normal.
- Uses the Rect class to draw a border around the RichTextArea subcomponent.
- Uses a Scroller component to add scroll bars to the RichTextArea subcomponent.

The skin class is defined in MXML, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark\myComponents\ModalTextSkin.mxml -->
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    minWidth="100" minHeight="25">
    <!-- Define ModalText as the host component of the skin. -->
    <fx:Metadata>
        <![CDATA[
            [HostComponent("myComponents.ModalText")]
        ]]>
    </fx:Metadata>
    <!-- Define the normal view state. -->
    <s:states>
        <s:State name="normal"/>
    </s:states>
    <!-- Define the border around the RichEditableText subcomponent. -->
    <s:Rect x="{myScroller.x}"
        width="{myScroller.width}"
        height="{myScroller.height}">
        <s:stroke>
            <s:SolidColorStroke color="0x686868" weight="1"/>
        </s:stroke>
    </s:Rect>
    <!--- Defines the appearance of the Button subcomponent. -->
    <s:Button id="modeButton"
        width="150"
        x="0"
        minHeight="25"/>

    <!--- Defines the appearance of the RichEditableText subcomponent. -->
    <s:Scroller id="myScroller"
        width="200"
        x="156">
        <s:RichEditableText id="textInput"
            minHeight="25"
            heightInLines="4"
            widthInChars="20"
            paddingLeft="4" paddingTop="4"
            paddingRight="4" paddingBottom="4"/>
    </s:Scroller>
</s:SparkSkin>
```

Creating the ModalText component

The following code implements the class definition for the ModalText component. The ModalText component is a composite component that contains a Button and a RichEditableText subcomponent:

```
package myComponents
{
    import flash.events.Event;
    import spark.components.Button;
    import spark.components.RichEditableText;
    import spark.components.supportClasses.SkinnableComponent;
    // ModalText dispatches a change event when the text of the
    // RichEditableText subcomponent changes.
    [Event(name="change", type="flash.events.Event")]

    /** a) Extend SkinnableComponent. */
    public class ModalText extends SkinnableComponent
    {
        /** b) Implement the constructor. */
        public function ModalText() {
            super();

            // Set the skinClass style to the name of the skin class.
            setStyle("skinClass", ModalTextSkin);
        }

        /** c) Define the skin parts for the Button
        *     and RichEditableText subcomponents. **/
        [SkinPart(required="true")]
        public var modeButton:Button;

        [SkinPart(required="true")]
        public var textInput:RichEditableText;

        /** d) Implement the commitProperties() method to handle the
        *     change to the ModalText.text property.
        *     Changes to the ModalText.text property are copied to
        *     the  RichEditableText subcomponent. */
        override protected function commitProperties():void {
            super.commitProperties();

            if (bTextChanged) {
                bTextChanged = false;
                textInput.text = _text;
            }
        }

        /** e) Implement the partAdded() method to
        *     initialize the Button and RichEditableText subcomponents. */
        override protected function partAdded(partName:String, instance:Object):void {
            super.partAdded(partName, instance);

            if (instance == textInput) {
                textInput.editable = false;
                textInput.text= _text;
                textInput.addEventListener("change", textInput_changeHandler);
            }
            if (instance == modeButton) {
                modeButton.label = "Toggle Editing Mode";
                modeButton.addEventListener("click", modeButton_clickHandler);
            }
        }
        /** f) Implement the partRemoved() method to remove the
        *     event listeners added by partAdded(). */
        override protected function partRemoved(partName:String, instance:Object):void {
            super.partRemoved(partName, instance);

            if (instance == textInput) {
                textInput.removeEventListener("change", textInput_changeHandler);
            }
            if (instance == modeButton) {
                textInput.removeEventListener("click", modeButton_clickHandler);
            }
        }
        /** g) Add methods, properties, and metadata.
        *     The general pattern for properties is to specify a
        *     private holder variable. */

        // Implement the ModalText.text property.
        private var _text:String = "ModalText";
        private var bTextChanged:Boolean = false;

        // Create a getter/setter pair for the text property.
        [Bindable]
        public function set text(t:String):void {
            _text = t;
            bTextChanged = true;
            invalidateProperties();
        }

        public function get text():String {
                return textInput.text;
        }

        // Dispatch a change event when the RichEditableText.text
        // property changes.
        private function textInput_changeHandler(eventObj:Event):void {
                dispatchEvent(new Event("change"));
        }

        // Handle the Button click event to toggle the
        // editting mode of the RichEditableText subcomponent.
        private function modeButton_clickHandler(eventObj:Event):void {
                textInput.editable = !textInput.editable;
        }
    }
}
```

Creating the ModalTextStates component

The following code example implements the class definition for the ModalTextStates component. The ModalTextStates component modifies the ModalText components shown in the previous section to add view states to the skin class. This control has all of the attributes of the ModalText class, and adds the following attributes:

- Uses the `textPlacement` property of the component to make the RichEditableText subcomponent appear on the right side or the left side of the component.
- Editing the `textPlacement` property of the control dispatches the `placementChanged` event.

- Uses the `textPlacement` property as the source for a data binding expression.
- Setting the `textPlacement` property so that the RichTextArea component appears on the right configures the skin class to use normal view state. Setting it to appear on the left uses the textLeft view state.
- Disabling editing of the RichTextArea component sets the view state of the skin class to normalDisabled or textLeftDisabled.

The following example uses the ModalTextStates component in an application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark/SparkMainModalTextStates.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
     <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
      <![CDATA[
        import flash.events.Event;

        private function placementChangedListener(event:Event):void {
          myEvent.text="placementChanged event occurred - textPlacement = "
              + myMT.textPlacement as String;
        }
      ]]>
    </fx:Script>
    <MyComp:ModalTextStates id="myMT"
        textPlacement="left"
        placementChanged="placementChangedListener(event);"/>
    <s:TextArea id="myEvent" width="50%" height="25"/>
    <s:Label text="Change Placement"/>
    <s:Button label="Set Text Placement Right"
        click="myMT.textPlacement='right';" />
    <s:Button label="Set Text Placement Left"
        click="myMT.textPlacement='left';"/>
</s:Application>
```

This applications sets the initial placement of the RichEditableText subcomponent to left. Use the buttons to switch the placement between right and left. When the placement changes, the event handler for the placementChanged event displays the current placement.

Creating the skin class for the ModalTextStates component

The skin class for the ModalTextStates component, ModalTextStatesSkin.mxml, defines the following view states to control the display of the component:

- normal The RichEditableText subcomponent is on the right, and editing is enabled.
- disabled The RichEditableText subcomponent is on the right, and editing is disabled.
- textLeft The RichEditableText subcomponent is on the left, and editing is enabled.
- textLeftDisabled The RichEditableText subcomponent is on the left, and editing is disabled.

Shown below is the skin definition:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark\myComponents\ModalTextStatesSkin.mxml -->
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    minWidth="100" minHeight="25">
    <!-- Define ModalTextState as the host component of the skin. -->
    <fx:Metadata>
        <![CDATA[
            [HostComponent("myComponents.ModalTextStates")]
        ]]>
    </fx:Metadata>
    <!-- Define the view states. -->
    <s:states>
        <s:State name="normal" />
        <s:State name="normalDisabled" stateGroups="disabledGroup"/>
        <s:State name="textLeft"/>
        <s:State name="textLeftDisabled" stateGroups="disabledGroup"/>
    </s:states>
    <!-- Define the border around the RichEditableText control. -->
    <s:Rect x="{myScroller.x}"
        width="{myScroller.width}"
        height="{myScroller.height}">
        <s:stroke>
            <s:SolidColorStroke color="0x686868" weight="1"/>
        </s:stroke>
    </s:Rect>
    <!--- Defines the appearance of the Button subcomponent. -->
    <s:Button id="modeButton"
        width="150"
        x="0"
        x.textLeft="206"
        x.textLeftDisabled="206"
        minHeight="25" height="100%"/>

    <!--- Defines the appearance of the RichEditableText subcomponent. -->
    <s:Scroller id="myScroller"
        width="200"
        x="156"
        x.textLeft="0"
        x.textLeftDisabled="0">
        <s:RichEditableText id="textInput"
            alpha="1.0" alpha.disabledGroup="0.5"
            minHeight="25"
            heightInLines="4"
            widthInChars="20"
            paddingLeft="4" paddingTop="4"
            paddingRight="4" paddingBottom="4"/>
    </s:Scroller>
</s:SparkSkin>
```

Creating the ModalTextStates component

Shown below is the code for the ModalTextStates component. The ModalTextStates class modifies the ModalText class in the following ways:

- Adds the implementation of the `ModalTextStates()` method to set the view state of the skin.

- Adds the implementation of the `textPlacement` property to set the placement of the RichEditableText subcomponent.

```
package myComponents
{
    import flash.events.Event;
    import spark.components.Button;
    import spark.components.RichEditableText;
    import spark.components.supportClasses.SkinnableComponent;
    // ModalText dispatches a change event when the text of the
    // RichEditableText subcomponent changes,
    // and a placementChanged event
    // when you change the textPlacement property of ModalText.
    [Event(name="change", type="flash.events.Event")]
    [Event(name="placementChanged", type="flash.events.Event")]
    // Define the skin states implemented by the skin class.
    [SkinState("normal")]
    [SkinState("normalDisabled")]
    [SkinState("textLeft")]
    [SkinState("textLeftDisabled")]

    /** a) Extend SkinnableComponent. */
    public class ModalTextStates extends SkinnableComponent
    {
        /** b) Implement the constructor. */
        public function ModalTextStates() {
            super();

            // Set the skin class.
            setStyle("skinClass", ModalTextStatesSkin);
        }

        /** C) Define the skin parts. */
        [SkinPart(required="true")]
        public var modeButton:Button;

        [SkinPart(required="true")]
        public var textInput:RichEditableText;

        /** d) Implement the commitProperties() method. */
        override protected function commitProperties():void {
            super.commitProperties();

            if (bTextChanged) {
                bTextChanged = false;
                textInput.text = _text;
            }
        }

        /** e) Implement the partAdded() method. */
        override protected function partAdded(partName:String, instance:Object):void {
            super.partAdded(partName, instance);

            if (instance == textInput) {
                textInput.editable = false;
                textInput.text= _text;
                textInput.addEventListener("change", textInput_changeHandler);
            }
            if (instance == modeButton) {
                modeButton.label = "Toggle Editing Mode";
                modeButton.addEventListener("click", modeButton_changeHandler);
            }
        }

        /** f) Implement the partRemoved() method. */
        override protected function partRemoved(partName:String, instance:Object):void {
            super.partRemoved(partName, instance);

            if (instance == textInput) {
                textInput.removeEventListener("change", textInput_changeHandler);
            }
            if (instance == modeButton) {
                textInput.removeEventListener("click", modeButton_changeHandler);
            }
        }
        /** g) Implement the getCurrentSkinState() method.
        *     The view state is determined by the _textPlacement property
        *     and the editable property of the RichEditableText subcomponent.*/
        override protected function getCurrentSkinState():String {
            var returnState:String = "normal";

            if (textInput.editable == true)
            {
                if (_textPlacement == "right") {
                    returnState = "normal";
                }
                else if (_textPlacement == "left") {
                    returnState = "textLeft";
                }
            }
            if (textInput.editable == false)
            {
                if (_textPlacement == "right") {
                    returnState = "normalDisabled";
                }
                else if (_textPlacement == "left") {
                    returnState = "textLeftDisabled";
                }
            }
            return returnState;
        }
        /** h) Add methods, properties, and metadata.
        *     The general pattern for properties is to specify a private
        *     holder variable. */

        // Define the text property.
        private var _text:String = "ModalText";
        private var bTextChanged:Boolean = false;

        // Then, create a getter/setter pair for the text property.
        [Bindable]
        public function set text(t:String):void {
            _text = t;
            bTextChanged = true;
```

```
            invalidateProperties();
        }

        public function get text():String {
            return textInput.text;
        }
        // Define the textPlacement property.
        private var _textPlacement:String = "right";

        // Create a getter/setter pair for the textPlacement property.
        // Dispatch the placementChanged event when it changes.
        [Bindable]
        public function set textPlacement(p:String):void {
            _textPlacement = p;
            invalidateSkinState();
            dispatchEvent(new Event("placementChanged"));
        }

        public function get textPlacement():String {
            return _textPlacement;
        }

        // Dispatch a change event when the RichEditableText.text
        // property changes.
        private function textInput_changeHandler(eventObj:Event):void {
            dispatchEvent(new Event("change"));
        }

        // Handle the Button click event to toggle the
        // editting mode of the RichEditableText subcomponent.
        private function modeButton_changeHandler(eventObj:Event):void {
            textInput.editable = !textInput.editable;
            invalidateSkinState();
        }
    }
}
```

Create advanced MX visual components in ActionScript

You can create advanced visual components for use in applications built with Adobe® Flex® .

About creating advanced MX components

Simple visual components are subclasses of existing Flex components that modify the appearance of the component by using skins or styles, or add new functionality to the component. For example, you add a new event type to a Button control, or modify the default styles or skins of a DataGrid control. For more information, see “Create simple visual components in ActionScript” on page 2433.

In advanced components, you typically perform the following actions:

- Modify the visual appearance or visual characteristics of an existing component.
- Create a composite component that encapsulates two or more components within it.
- Create a component by creating a subclass of the UIComponent class.

You usually create a component as a subclass of an existing class. For example, to create a component that is based on the Button control, you create a subclass of the mx.controls.Button class. To make your own component, you create a subclass of the mx.core.UIComponent class.

About overriding protected UIComponent methods for MX components

All Flex visual components are subclasses of the UIComponent class. Therefore, visual components inherit the methods, properties, events, styles, and effects defined by the UIComponent class.

To create an advanced visual component, you must implement a class constructor. Also, you optionally override one or more of the following protected methods of the UIComponent class:

UIComponent method	Description
`commitProperties()`	Commits any changes to component properties, either to make the changes occur at the same time or to ensure that properties are set in a specific order. For more information, see "Implementing the commitProperties() method for MX components" on page 2483.
`createChildren()`	Creates any child components of the component. For example, the ComboBox control contains a TextInput control and a Button control as child components. For more information, see "Implementing the createChildren() method for MX components" on page 2482.
`layoutChrome()`	Defines the border area around the container for subclasses of the Container class. For more information, see "Implementing the layoutChrome() method for MX components" on page 2488.
`measure()`	Sets the default size and default minimum size of the component. For more information, see "Implementing the measure() method for MX components" on page 2485.
`updateDisplayList()`	Sizes and positions the children of the component on the screen based on all previous property and style settings, and draws any skins or graphic elements used by the component. The parent container for the component determines the size of the component itself. For more information, see "Implementing the updateDisplayList() method for MX components" on page 2489.

Component users do not call these methods directly; Flex calls them as part of the initialization process of creating a component, or when other method calls occur. For more information, see "About the component instantiation life cycle for MX components" on page 2477.

About the invalidation methods for MX components

During the lifetime of a component, your application might modify the component by changing its size or position, modifying a property that controls its display, or modifying a style or skin property of the component. For example, you might change the font size of the text displayed in a component. As part of changing the font size, the component's size might also change, which requires Flex to update the layout of the application. The layout operation might require Flex to invoke the `commitProperties()`, `measure()`, `layoutChrome()`, and the `updateDisplayList()` methods of your component.

Your application can programmatically change the font size of a component much faster than Flex can update the layout of an application. Therefore, you should only want to update the layout after you are sure that you've determined the final value of the font size.

In another scenario, when you set multiple properties of a component, such as the `label` and `icon` properties of a Button control, you want the `commitProperties()`, `measure()`, and `updateDisplayList()` methods to execute only once, after all properties are set. You do not want these methods to execute when you set the `label` property, and then execute again when you set the `icon` property.

Also, several components might change their font size at the same time. Rather than updating the application layout after each component changes its font size, you want Flex to coordinate the layout operation to eliminate any redundant processing.

Flex uses an invalidation mechanism to synchronize modifications to components. Flex implements the invalidation mechanism as a set of methods that you call to signal that something about the component has changed and requires Flex to call the component's `commitProperties()`, `measure()`, `layoutChrome()`, or `updateDisplayList()` methods.

The following table describes the invalidation methods:

Invalidation method	**Description**
`invalidateProperties()`	Marks a component so that its `commitProperties()` method gets called during the next screen update.
`invalidateSize()`	Marks a component so that its `measure()` method gets called during the next screen update.
`invalidateDisplayList()`	Marks a component so that its `layoutChrome()` and `updateDisplayList()` methods get called during the next screen update.

When a component calls an invalidation method, it signals to Flex that the component must be updated. When multiple components call invalidation methods, Flex coordinates updates so that they all occur together during the next screen update.

Typically, component users do not call the invalidation methods directly. Instead, they are called by the component's setter methods, or by any other methods of a component class as necessary. For more information and examples, see "Implementing the commitProperties() method for MX components" on page 2483.

About the component instantiation life cycle for MX components

The component instantiation life cycle describes the sequence of steps that occur when you create a component object from a component class. As part of that life cycle, Flex automatically calls component methods, dispatches events, and makes the component visible.

The following example creates a Button control in ActionScript and adds it to a container:

```
// Create a Box container.
var boxContainer:Box = new Box();
// Configure the Box container.

// Create a Button control.
var b:Button = new Button()
// Configure the button control.
b.label = "Submit";
...
// Add the Button control to the Box container.
boxContainer.addChild(b);
```

The following steps show what occurs when you execute the code to create the Button control, and add the control to the Box container:

1 You call the component's constructor, as the following code shows:

```
// Create a Button control.
var b:Button = new Button()
```

2 You configure the component by setting its properties, as the following code shows:

```
// Configure the button control.
b.label = "Submit";
```

Component setter methods might call the `invalidateProperties()`, `invalidateSize()`, or `invalidateDisplayList()` methods.

3 You call the `addChild()` method to add the component to its parent, as the following code shows:

```
// Add the Button control to the Box container.
boxContainer.addChild(b);
```

Flex then performs the following actions:

4 Sets the `parent` property for the component to reference its parent container.

5 Computes the style settings for the component.

6 Dispatches the `preinitialize` event on the component.

7 Calls the component's `createChildren()` method.

8 Calls the `invalidateProperties()`, `invalidateSize()`, and `invalidateDisplayList()` methods to trigger later calls to the `commitProperties()`, `measure()`, or `updateDisplayList()` methods during the next `render` event.

The only exception to this rule is that Flex does not call the `measure()` method when the user sets the height and width of the component.

9 Dispatches the `initialize` event on the component. At this time, all of the component's children are initialized, but the component has not been sized or processed for layout. You can use this event to perform additional processing of the component before it is laid out.

10 Dispatches the `childAdd` event on the parent container.

11 Dispatches the `initialize` event on the parent container.

12 During the next `render` event, Flex performs the following actions:

a Calls the component's `commitProperties()` method.

b Calls the component's `measure()` method.

c Calls the component's `layoutChrome()` method.

d Calls the component's `updateDisplayList()` method.

e If steps 13 and 14 are not required, which is the most common scenario, dispatches the `updateComplete` event on the component.

13 Flex dispatches additional `render` events if the `commitProperties()`, `measure()`, or `updateDisplayList()` methods call the `invalidateProperties()`, `invalidateSize()`, or `invalidateDisplayList()` methods.

14 After the last `render` event occurs, Flex performs the following actions:

a Makes the component visible by setting the `visible` property to `true`.

b Dispatches the `creationComplete` event on the component. The component is sized and processed for layout. This event is only dispatched once when the component is created.

c Dispatches the `updateComplete` event on the component. Flex dispatches additional `updateComplete` events whenever the layout, position, size, or other visual characteristic of the component changes and the component is updated for display.

Most of the work for configuring a component occurs when you add the component to a container by using the `addChild()` method. That is because until you add the component to a container, Flex cannot determine its size, set inheriting style properties, or draw it on the screen.

You can also define your application in MXML, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <mx:Box>
        <mx:Button label="Submit"/>
    </mx:Box>
</s:Application>
```

The sequence of steps that Flex executes when creating a component in MXML are equivalent to the steps described for ActionScript.

You can remove a component from a container by using the `removeChild()` method. If there are no references to the component, it is eventually deleted from memory by the garbage collection mechanism of Adobe® Flash® Player or Adobe® AIR™.

About the steps for creating a component for MX components

When you implement a component, you override component methods, define new properties, dispatch new events, or perform any other customizations required by your application.

To implement your component, follow these general steps:

1 If necessary, create any skins for the component.

2 Create an ActionScript class file.

  a Extend one of the base classes, such as UIComponent or another component class.

  b Specify properties that the user can set by using an MXML tag property.

  c Embed any graphic and skin files.

  d Implement the constructor.

  e Implement the UIComponent.createChildren() method.

  f Implement the UIComponent.commitProperties() method.

  g Implement the UIComponent.measure() method.

  h Implement the UIComponent.layoutChrome() method.

  i Implement the UIComponent.updateDisplayList() method.

  j Add properties, methods, styles, events, and metadata.

3 Deploy the component as an ActionScript file or as a SWC file.

For more information about MXML tag properties and embedding graphic and skin files, see “Create simple visual components in ActionScript” on page 2433.

You do not have to override all component methods to define a new component. You only override the methods required to implement the functionality of your component. If you create a subclass of an existing component, such as Button control or VBox container, you must implement the methods necessary for you to add any new functionality to the component.

For example, you can implement a custom Button control that uses a new mechanism for defining its default size. In that case, you only need to override the `measure()` method. For an example, see "Implementing the measure() method for MX components" on page 2485.

Or you might implement a new subclass of the VBox container. Your new subclass uses all of the existing sizing logic of the VBox class, but changes the layout logic of the class to lay out the container children from the bottom of the container to the top, rather than from the top down. In this case, you only need to override the `updateDisplayList()` method. For an example, see "Implementing the updateDisplayList() method for MX components" on page 2489.

About interfaces

Flex uses interfaces to divide the basic functionality of components into discrete elements so that they can be implemented piece by piece. For example, to make your component focusable, it must implement the IFocusable interface; to let it participate in the layout process, it must implement ILayoutClient interface.

To simplify the use of interfaces, the UIComponent class implements all of the interfaces defined in the following table, except for the IFocusManagerComponent and IToolTipManagerClient interfaces. However, many subclasses of UIComponent implement the IFocusManagerComponent and IToolTipManagerClient interfaces.

Therefore, if you create a subclass of the class or subclass of UIComponent, you do not have to implement these interfaces. But, if you create a component that is not a subclass of UIComponent, and you want to use that component in Flex, you might have to implement one or more of these interfaces.

***Note:** For Flex, Adobe recommends that all of your components extend the UIComponent class or a class that extends UIComponent.*

The following table lists the main interfaces implemented by Flex components:

Interface	Use
IAdvancedStyleClient	Indicates that the component supports the advanced style subsystem.
IAutomationObject	Indicates that a component is an object within the automation object hierarchy.
IChildList	Indicates the number of children in a container.
IConstraintClient	Indicates that the component support layout constraints.
IDeferredInstantiationUIComponent	Indicates that a component or object can effect deferred instantiation.
IFlexDisplayObject	Specifies the interface for skin elements.
IFlexModule	indicates that the component can be used with module factories
IInvalidating	Indicates that a component or object can use the invalidation mechanism to perform delayed, rather than immediate, property commitment, measurement, and drawing or layout.
ILayoutManagerClient	Indicates that a component or object can participate in the LayoutManager's commit, measure, and update sequence.
IPropertyChangeNotifier	Indicates that a component supports a specialized form of event propagation.
IRepeaterClient	Indicates that a component or object can be used with the Repeater class.
IStateClient	Indicates that the component supports view states.

Interface	Use
IToolTipManagerClient	Indicates that a component has a `toolTip` property, and therefore is monitored by the ToolTipManager.
IUIComponent	Defines the basic set of APIs that you must implement in order to be a child of layout containers and lists.
IValidatorListener	Indicates that a component can listen for validation events, and therefore show a validation state, such as a red border and error tooltips.
IVisualElement	Indicates that the component can be laid out and displayed in a Spark application.

Implementing the component

When you create a custom component in ActionScript, you have to override the methods of the UIComponent class. You implement the basic component structure, the constructor, and the `createChildren()`, `commitProperties()`, `measure()`, `layoutChrome()`, and `updateDisplayList()` methods.

Basic component structure

The following example shows the basic structure of a Flex component:

```
package myComponents
{
    public class MyComponent extends UIComponent
    {
        ....
    }
}
```

You must define your ActionScript custom components within a package. The package reflects the directory location of your component within the directory structure of your application.

The class definition of your component must be prefixed by the `public` keyword. A file that contains a class definition can have one, and only one, public class definition, although it can have additional internal class definitions. Place any internal class definitions at the bottom of your source file below the closing curly brace of the package definition.

Implementing the constructor

Your ActionScript class should define a public constructor method for a class that is a subclass of the UIComponent class, or a subclass of any child of the UIComponent class. The constructor has the following characteristics:

- No return type
- Should be declared public
- No arguments
- Calls the `super()` method to invoke the superclass' constructor

Each class can contain only one constructor method; ActionScript does not support overloaded constructor methods. For more information, see "Defining the constructor" on page 2366.

Use the constructor to set the initial values of class properties. For example, you can set default values for properties and styles, or initialize data structures, such as Arrays.

Do not create child display objects in the constructor; you should use it only for setting initial properties of the component. If your component creates child components, create them in the `createChildren()` method.

Implementing the createChildren() method for MX components

A component that creates other components or visual objects within it is called a *composite component*. For example, the Flex ComboBox control contains a TextInput control to define the text area of the ComboBox, and a Button control to define the ComboBox arrow. Components implement the `createChildren()` method to create child objects (such as other components) in the component.

You do not call the createChildren() method directly; Flex calls it when the call to the `addChild()` method occurs to add the component to its parent. Notice that the `createChildren()` method has no invalidation method, which means that you do not have to call it a second time after the component is added to its parent.

For example, you might define a new component that consists of a Button control and a TextArea control, where the Button control enables and disables user input to the TextArea control. The following example creates the TextArea and Button controls:

```
// Declare two variables for the component children.
private var text_mc:TextArea;
private var mode_mc:Button;

override protected function createChildren():void {

    // Call the createChildren() method of the superclass.
    super.createChildren();

    // Test for the existence of the children before creating them.
    // This is optional, but do this so a subclass can create a different
    // child.
    if (!text_mc) {
        text_mc = new TextArea();
        text_mc.explicitWidth = 80;
        text_mc.editable = false;
        text_mc.addEventListener("change", handleChangeEvent);
        // Add the child component to the custom component.
        addChild(text_mc);
    }

    // Test for the existence of the children before creating them.
    if (!mode_mc) {
        mode_mc = new Button();
        mode_mc.label = "Toggle Editing";
        mode_mc.addEventListener("click", handleClickEvent);
        // Add the child component to the custom component.
        addChild(mode_mc);
    }
}
```

Notice in this example that the `createChildren()` method calls the `addChild()` method to add the child component. You must call the `addChild()` method for each child object.

After you create a child component, you can use properties of the child component to define its characteristics. In this example, you create the Button and TextArea controls, initialize them, and register event listeners for them. You could also apply skins to the child components. For a complete example, see "Example: Creating a composite MX component" on page 2493.

Implementing the commitProperties() method for MX components

You use the commitProperties() method to coordinate modifications to component properties. Most often, you use it with properties that affect how a component appears on the screen.

Flex schedules a call to the `commitProperties()` method when a call to the `invalidateProperties()` method occurs. The `commitProperties()` method executes during the next `render` event after a call to the `invalidateProperties()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateProperties()` method.

Calls to the `commitProperties()` method occur before calls to the `measure()` method. This lets you set property values that the `measure()` method might use.

The typical pattern for defining component properties is to define the properties by using getter and setter methods, as the following example shows:

```
// Define a private variable for the alignText property.
private var _alignText:String = "right";

// Define a flag to indicate when the _alignText property changes.
private var bAlignTextChanged:Boolean = false;

// Define getter and setter methods for the property.
public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;

    // Trigger the commitProperties(), measure(), and updateDisplayList()
    // methods as necessary.
    // In this case, you do not need to remeasure the component.
    invalidateProperties();
    invalidateDisplayList();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // Handle alignment change
    }
}
```

As you can see in this example, the setter method modifies the property, calls the `invalidateProperties()` and `invalidateDisplayList()` methods, and then returns. The setter itself does not perform any calculations based on the new property value. This design lets the setter method return quickly, and leaves any processing of the new value to the `commitProperties()` method.

Changing the alignment of text in a control does not necessarily change the control’s size. However, if it does, include a call to the `invalidateSize()` method to trigger the `measure()` method.

The main advantages of using the `commitProperties()` method are the following:

- To coordinate the modifications of multiple properties so that the modifications occur synchronously.

  For example, you might define multiple properties that control the text displayed by the component, such as the alignment of the text within the component. A change to either the text or the alignment property requires Flex to update the appearance of the component. However, if you modify both the text and the alignment, you want Flex to perform any calculations for sizing or positioning the component once, when the screen updates.

  Therefore, you use the `commitProperties()` method to calculate any values based on the relationship of multiple component properties. By coordinating the property changes in the `commitProperties()` method, you can reduce unnecessary processing overhead.

- To coordinate multiple modifications to the same property.

  You do not necessarily want to perform a complex calculation every time a user updates a component property. For example, users modify the `icon` property of the Button control to change the image displayed in the button. Calculating the label position based on the presence or size of an icon can be a computationally expensive operation that you want to perform only when necessary.

  To avoid this behavior, you use the `commitProperties()` method to perform the calculations. Flex calls the `commitProperties()` method when it updates the display. That means you perform the calculations once when Flex updates the screen, regardless of the number of times the property changed between screen updates.

The following example shows how you can handle two related properties in the `commitProperties()` method:

```
// Define a private variable for the text property.
private var _text:String = "ModalText";
private var bTextChanged:Boolean = false;

// Define the getter method.
public function get text():String {
        return _text;
}

//Define the setter method to call invalidateProperties()
// when the property changes.
public function set text(t:String):void {
    _text = t;
    bTextChanged = true;
    invalidateProperties();
    // Changing the text causes the control to recalculate its default size.
    invalidateSize();
    invalidateDisplayList();
}

// Define a private variable for the alignText property.
private var _alignText:String = "right";
private var bAlignTextChanged:Boolean = false;

public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;
    invalidateProperties();
    invalidateDisplayList();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flags indicate a change to both properties.
    if (bTextChanged && bAlignTextChanged) {
        // Reset flags.
        bTextChanged = false;
        bAlignTextChanged = false;

        // Handle case where both properties changed.
    }

    // Check whether the flag indicates a change to the text property.
    if (bTextChanged) {
        // Reset flag.
        bTextChanged = false;

        // Handle text change.
    }

    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // Handle alignment change.
    }
}
```

Implementing the measure() method for MX components

The measure() method sets the default component size, in pixels, and optionally sets the component's default minimum size.

Flex schedules a call to the `measure()` method when a call to the `invalidateSize()` method occurs. The `measure()` method executes during the next `render` event after a call to the `invalidateSize()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateSize()` method.

When you set a specific height and width of a component, Flex does not call the `measure()` method, even if you explicitly call the `invalidateSize()` method. That is, Flex calls the `measure()` method only if the `explicitWidth` property or the `explicitHeight` property of the component is `NaN`.

In the following example, because you explicitly set the size of the Button control, Flex does not call the `Button.measure()` method:

```
<mx:Button height="10" width="10"/>
```

In a subclass of an existing component, you might implement the `measure()` method only if you are performing an action that requires modification to the default sizing rules defined in the superclass. Therefore, to set a new default size, or perform calculations at run time to determine component sizing rules, implement the `measure()` method.

You set the following properties in the `measure()` method to specify the default size:

Properties	Description
`measuredHeightmea suredWidth`	Specifies the default height and width of the component, in pixels. These properties are set to 0 until the `measure()` method executes. Although you can leave them set to 0, it makes the component invisible by default.
`measuredMinHeight measuredMinWidth`	Specifies the default minimum height and minimum width of the component, in pixels. Flex cannot set the size of a component smaller than its specified minimum size.

The `measure()` method only sets the default size of the component. In the `updateDisplayList()` method, the parent container of the component passes to it its actual size, which may be different than the default size.

Component users can also override the default size settings in an application by using the component in the following ways:

- Setting the `explicitHeight` and `exlicitWidth` properties
- Setting the `width` and `height` properties
- Setting the `percentHeight` and `percentWidth` properties

For example, you can define a Button control with a default size of 100 pixels wide and 50 pixels tall, and a default minimum size of 50 pixels by 25 pixels, as the following example shows:

```
package myComponents
{
    // asAdvanced/myComponents/BlueButton.as
    import mx.controls.Button;
    public class BlueButton extends Button {

        public function BlueButton() {
            super();
        }
        override protected function measure():void {
            super.measure();

            measuredWidth=100;
            measuredMinWidth=50;
            measuredHeight=50;
            measuredMinHeight=25;
        }
    }
}
```

The following application uses this button in an application:

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainBlueButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*" >
    <mx:VBox>
        <MyComp:BlueButton/>
        <mx:Button/>
    </mx:VBox>
</s:Application>
```

In the absence of any other sizing constraints on the button, the VBox container uses the default size and default minimum size of the button to calculate its size at run time. For information on the rules for sizing a component, see "Introduction to containers" on page 326.

You can override the default size settings in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- asAdvanced/MainBlueButtonResize.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*" >
    <mx:VBox>
        <MyComp:BlueButton width="50%"/>
        <mx:Button/>
    </mx:VBox>
</s:Application>
```

In this example, you specify that the width of the button is 50% of the width of the VBox container. When 50% of the width of the container is smaller than the minimum width of the button, the button uses its minimum width.

Calculating default sizes

The example in "Implementing the measure() method for MX components" on page 2485 uses static values for the default size and default minimum size of a component. Some Flex components use static sizes. For example, the TextArea control has a default size of 100 pixels wide by 44 pixels high, regardless of the text it contains. If the text is larger than the TextArea control, the control displays scroll bars.

Often, you set the default size based on characteristics of the component or information passed to the component. For example, the Button control's measure() method examines its label text, margin settings, and font characteristics to determine the control's default size.

In the following example, you override the `measure()` method of the TextArea control so that it examines the text passed to the control, and calculates the default size of the TextArea control to display the entire text string in a single line:

```
package myComponents
{
    // asAdvanced/myComponents/MyTextArea.as
    import mx.controls.TextArea;
    import flash.text.TextLineMetrics;
    public class MyTextArea extends TextArea
    {
        public function MyTextArea() {
            super();
        }

        // The default size is the size of the text plus a 10 pixel margin.
        override protected function measure():void {
            super.measure();
            // Calculate the default size of the control based on the
            // contents of the TextArea.text property.
            var lineMetrics:TextLineMetrics = measureText(text);
            // Add a 10 pixel border area around the text.
            measuredWidth = measuredMinWidth = lineMetrics.width + 10;
            measuredHeight = measuredMinHeight = lineMetrics.height + 10;
        }
    }
}
```

For text strings that are longer than the display area of your application, you can add logic to increase the height of the TextArea control to display the text on multiple lines. The following application uses this component:

```
<?xml version="1.0"?>
<!-- asAdvanced/MainMyTextArea.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*"
    width="1000">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <MyComp:MyTextArea id="myTA" text="This is a long text strring that would normally cause
a TextArea control to display scroll bars. But, the custom MyTextArea control calcualtes its
default size based on the text size."/>

    <mx:TextArea id="flexTA" text="This is a long text strring that would normally cause a
TextArea control to display scroll bars. But, the custom MyTextArea control calcualtes its
default size based on the text size."/>
</s:Application>
```

Implementing the layoutChrome() method for MX components

The Container class, and some subclasses of the Container class, use the layoutChrome() method to define the border area around the container.

Flex schedules a call to the `layoutChrome()` method when a call to the `invalidateDisplayList()` method occurs. The `layoutChrome()` method executes during the next `render` event after a call to the `invalidateDisplayList()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateDisplayList()` method.

Typically, you use the RectangularBorder class to define the border area of a container. For example, you can create the RectangularBorder object, and add it as a child of the component in your override of the `createChildren()` method.

When you create a subclass of the Container class, you can use the `createChildren()` method to create the content children of the container; the content children are the child components that appear within the container. You then use `updateDisplayList()` to position the content children.

You typically use the `layoutChrome()` method to define and position the border area of the container, and any additional elements that you want to appear in the border area. For example, the Panel container uses the `layoutChrome()` method to define the title area of the panel container, including the title text and close button.

The primary reason for dividing the handling of the content area of a container from its border area is to handle the situation when the `Container.autoLayout` property is set to `false`. When the `autoLayout` property is set to `true`, measurement and layout of the container and of its children are done whenever the position or size of a container child changes. The default value is `true`.

When the `autoLayout` property is set to `false`, measurement and layout are done only once, when children are added to or removed from the container. However, Flex executes the `layoutChrome()` method in both cases. Therefore, the container can still update its border area even when the `autoLayout` property is set to `false`.

Implementing the updateDisplayList() method for MX components

The updateDisplayList() method sizes and positions the children of your component based on all previous property and style settings, and draws any skins or graphic elements that the component uses. The parent container for the component determines the size of the component itself.

A component does not appear on the screen until its `updateDisplayList()` method gets called. Flex schedules a call to the `updateDisplayList()` method when a call to the `invalidateDisplayList()` method occurs. The `updateDisplayList()` method executes during the next `render` event after a call to the `invalidateDisplayList()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateDisplayList()` method.

The main uses of the `updateDisplayList()` method are the following:

- To set the size and position of the elements of the component for display.

  Many components are made up of one or more child components, or have properties that control the display of information in the component. For example, the Button control lets you specify an optional icon, and use the `labelPlacement` property to specify where the button text appears relative to the icon.

  The `Button.updateDisplayList()` method uses the settings of the `icon` and `labelPlacement` properties to control the display of the button.

  For containers that have child controls, the `updateDisplayList()` method controls how those child components are positioned. For example, the `updateDisplayList()` method on the HBox container positions its children from left to right in a single row; the `updateDisplayList()` method for a VBox container positions its children from top to bottom in a single column.

  To size components in the `updateDisplayList()` method, you use the `setActualSize()` method, not the sizing properties, such as `width` and `height`. To position a component, use the `move()` method, not the `x` and `y` properties.

- To draw any visual elements necessary for the component.

  Components support many types of visual elements such as skins, styles, and borders. Within the `updateDisplayList()` method, you can add these visual elements, use the Flash drawing APIs, and perform additional control over the visual display of your component.

The `updateDisplayList()` method has the following signature:

```
protected function updateDisplayList(unscaledWidth:Number,
    unscaledHeight:Number):void
```

The properties have the following values:

unscaledWidth Specifies the width of the component, in pixels, in the component’s coordinates, regardless of the value of the scaleX property of the component. This is the width of the component as determined by its parent container.

unscaledHeight Specifies the height of the component, in pixels, in the component’s coordinates, regardless of the value of the `scaleY` property of the component. This is the height of the component as determined by its parent container.

Scaling occurs in Flash Player or AIR, after `updateDisplayList()` executes. For example, a component with an `unscaledHeight` value of 100, and with a `scaleY` property of 2.0, appears 200 pixels high in Flash Player or AIR.

Overriding the layout mechanism of the VBox container

The VBox container lays out its children from the top of the container to the bottom, in the order in which the children are added to the container. The following example overrides the updateDisplayList() method, which causes the VBox container to layout its children from the bottom of the container to the top:

```
package myComponents
{
    // asAdvanced/myComponents/BottomUpVBox.as
    import mx.containers.VBox;
    import mx.core.EdgeMetrics;
    import mx.core.UIComponent;
    public class BottomUpVBox extends VBox
    {

        public function BottomUpVBox() {
            super();
        }

        override protected function updateDisplayList(unscaledWidth:Number,
            unscaledHeight:Number):void {
            super.updateDisplayList(unscaledWidth, unscaledHeight);

            // Get information about the container border area.
            // The usable area of the container for its children is the
            // container size, minus any border areas.
            var vm:EdgeMetrics = viewMetricsAndPadding;
            // Get the setting for the vertical gap between children.
            var gap:Number = getStyle("verticalGap");

            // Determine the y coordinate of the bottom of the usable area
            // of the VBox.
            var yOfComp:Number = unscaledHeight-vm.bottom;

            // Temp variable for a container child.
            var obj:UIComponent;

            for (var i:int = 0; i < numChildren; i++)
            {
                // Get the first container child.
                obj = UIComponent(getChildAt(i));

                // Determine the y coordinate of the child.
                yOfComp = yOfComp - obj.height;

                // Set the x and y coordinate of the child.
                // Note that you do not change the x coordinate.
                obj.move(obj.x, yOfComp);

                // Save the y coordinate of the child,
                // plus the vertical gap between children.
                // This is used to calculate the coordinate
                // of the next child.
                yOfComp = yOfComp - gap;
            }
        }
    }
}
```

In this example, you use the `UIComponent.move()` method to set the position of each child in the container. You can also use the `UIComponent.x` and `UIComponent.y` properties to set these coordinates. The difference is that the `move()` method changes the location of the component and then dispatches a `move` event when you call the method immediately; setting the `x` and `y` properties changes the location of the component and dispatches the event on the next screen update.

The following application uses this component:

```
<?xml version="1.0"?>
<!-- asAdvanced/MainBottomVBox.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*" >
  <MyComp:BottomUpVBox>
    <mx:Label text="Label 1"/>
    <mx:Button label="Button 1"/>
    <mx:Label text="Label 2"/>
    <mx:Button label="Button 2"/>

    <mx:Label text="Label 3"/>
    <mx:Button label="Button 3"/>
    <mx:Label text="Label 4"/>
    <mx:Button label="Button 4"/>
  </MyComp:BottomUpVBox>
</s:Application>
```

Drawing graphics in your component

Every Flex component is a subclass of the Flash Sprite class, and therefore inherits the `Sprite.graphics` property. The `Sprite.graphics` property specifies a Graphics object that you can use to add vector drawings to your component.

For example, in the `updateDisplayList()` method, you can use methods of the Graphics class to draw borders, rules, and other graphical elements:

```
override protected function updateDisplayList(unscaledWidth:Number, unscaledHeight:Number):void {

    super.updateDisplayList(unscaledWidth, unscaledHeight);

    // Draw a simple border around the child components.
    graphics.lineStyle(1, 0x000000, 1.0);
    graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);
}
```

Making components accessible

A growing requirement for web content is that it should be accessible to people who have disabilities. Visually impaired people can use the visual content in Flash applications by using screen reader software, which provides an audio description of the material on the screen.

When you create a component, you can include ActionScript that enables the component and a screen reader for audio communication. When developers use your component to build an application in Flash, they use the Accessibility panel to configure each component instance.

Flash includes the following accessibility features:

- Custom focus navigation
- Custom keyboard shortcuts
- Screen-based documents and the screen authoring environment
- An Accessibility class

To enable accessibility in your component, add the following line to your component's class file:

```
mx.accessibility.ComponentName.enableAccessibility();
```

For example, the following line enables accessibility for the MyButton component:

```
mx.accessibility.MyButton.enableAccessibility();
```

For additional information about accessibility, see "Accessible applications" on page 2122.

Adding version numbers

When releasing components, you can define a version number. This lets developers know whether they should upgrade, and helps with technical support issues. When you set a component's version number, use the static variable `version`, as the following example shows:

```
static var version:String = "1.0.0.42";
```

***Note:** Flex does not use or interpret the value of the `version` property.*

If you create many components as part of a component package, you can include the version number in an external file. That way, you update the version number in only one place. For example, the following code imports the contents of an external file that stores the version number in one place:

```
include "../myPackage/ComponentVersion.as"
```

The contents of the ComponentVersion.as file are identical to the previous variable declaration, as the following example shows:

```
static var version:String = "1.0.0.42";
```

Best practices when designing a component

Use the following practices when you design a component:

- Keep the file size as small as possible.
- Make your component as reusable as possible by generalizing functionality.
- Use the Border class rather than graphical elements to draw borders around objects.
- Assume an initial state. Because style properties are on the object, you can set initial settings for styles and properties so your initialization code does not have to set them when the object is constructed, unless the user overrides the default state.

Example: Creating a composite MX component

Composite components are components that contain multiple components. They might be graphical assets or a combination of graphical assets and component classes. For example, you can create a component that includes a button and a text field, or a component that includes a button, a text field, and a validator.

When you create composite components, you should instantiate the controls inside the component's class file. Assuming that some of these controls have graphical assets, you must plan the layout of the controls that you are including, and set properties such as default values in your class file. You must also ensure that you import all the necessary classes that the composite component uses.

Because the class extends one of the base classes, such as UIComponent, and not a controls class like Button, you must instantiate each of the controls as children of the custom component and arrange them on the screen.

Properties of the individual controls are not accessible from the MXML author's environment unless you design your class to allow this. For example, if you create a component that extends the UIComponent class and uses a Button and a TextArea component, you cannot set the Button control's label text in the MXML tag because you do not directly extend the Button class.

Creating the component

This example component, called ModalText and defined in the file ModalText.as, combines a Button control and a TextArea control. You use the Button control to enable or disable text input in the TextArea control.

Defining event listeners for composite components

Custom components implement the `createChildren()` method to create children of the component, as the following example shows:

```
override protected function createChildren():void {
    super.createChildren();

    // Create and initialize the TextArea control.
    if (!text_mc) {
        text_mc = new TextArea();
        ...
        text_mc.addEventListener("change", handleChangeEvent);
        addChild(text_mc);
    }

    // Create and initialize the Button control.
    if (!mode_mc) {
        mode_mc = new Button();
        ...
        mode_mc.addEventListener("click", handleClickEvent);
        addChild(mode_mc);
    }
}
```

The `createChildren()` method also contains a call to the `addEventListener()` method to register an event listener for the `change` event generated by the TextArea control, and for the `click` event for the Button control. These event listeners are defined within the ModalText class, as the following example shows:

```
// Handle events that are dispatched by the children.
private function handleChangeEvent(eventObj:Event):void {
        dispatchEvent(new Event("change"));
}

// Handle events that are dispatched by the children.
private function handleClickEvent(eventObj:Event):void {
        text_mc.editable = !text_mc.editable;
}
```

You can handle an event dispatched by a child of a composite component in the component. In this example, the event listener for the Button control's `click` event is defined in the class definition to toggle the `editable` property of the TextArea control.

However, if a child component dispatches an event, and you want that opportunity to handle the event outside of the component, you must add logic to your custom component to propagate the event. Notice that the event listener for the `change` event for the TextArea control propagates the event. This lets you handle the event in your application, as the following example shows:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*">

    <fx:Script>
        <![CDATA[

            import flash.events.Event;

            function handleText(eventObj:Event)
            {
                ...
            }
        ]]>
    </fx:Script>

    <MyComp:ModalText change="handleText(event);"/>
</s:Application>
```

Creating the ModalText component

The following code example implements the class definition for the ModalText component. The ModalText component is a composite component that contains a Button control and a TextArea control. This control has the following attributes:

- You cannot edit the TextArea control by default.
- You click the Button control to toggle editing of the TextArea control.
- You use the `textPlacement` property of the control to make the TextArea appear on the right side or the left side of the control.
- Editing the `textPlacement` property of the control dispatches the `placementChanged` event.
- You use the `text` property to programmatically write content to the TextArea control.
- Editing the `text` property of the control dispatches the `textChanged` event.
- Editing the text in the TextArea control dispatches the `change` event.
- You can use both the `textPlacement` property or the `text` property as the source for a data binding expression.
- You can optionally use skins for the up, down, and over states of the Button control.

The following is an example MXML file that uses the ModalText control and sets the `textPlacement` property to `left`:

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainModalText.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*" >
    <MyComp:ModalText textPlacement="left" height="40"/>
</s:Application>
```

You can handle the `placementChanged` event to determine when the `ModalText.textPlacement` property is modified, as the following example shows:

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainModalTextEvent.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Script>
      <![CDATA[
        import flash.events.Event;

        private function placementChangedListener(event:Event):void {
          myEvent.text="placementChanged event occurred - textPlacement = "
              + myMT.textPlacement as String;
        }
      ]]>
    </fx:Script>
    <MyComp:ModalText id="myMT"
        textPlacement="left"
        height="40"
        placementChanged="placementChangedListener(event);"/>
    <mx:TextArea id="myEvent" width="50%"/>
    <mx:Label text="Change Placement" />
    <mx:Button label="Set Text Placement Right"
        click="myMT.textPlacement='right';" />
    <mx:Button label="Set Text Placement Left"
        click="myMT.textPlacement='left';" />
</s:Application>
```

The following example shows the ModalText.as file that defines this control:

```
package myComponents
{
    // asAdvanced/myComponents/ModalText.as
    // Import all necessary classes.
    import mx.core.UIComponent;
    import mx.controls.Button;
    import mx.controls.TextArea;
    import flash.events.Event;
    import flash.text.TextLineMetrics;
    // ModalText dispatches a change event when the text of the child
    // TextArea control changes, a textChanged event when you set the text
    // property of ModalText, and a placementChanged event
    // when you change the textPlacement property of ModalText.
    [Event(name="change", type="flash.events.Event")]
    [Event(name="textChanged", type="flash.events.Event")]
    [Event(name="placementChanged", type="flash.events.Event")]

    /*** a) Extend UIComponent. ***/
    public class ModalText extends UIComponent {
        /*** b) Implement the class constructor. ***/
        public function ModalText() {
            super();
        }
        /*** c) Define variables for the two child components. ***/
```

```
// Declare two variables for the component children.
private var text_mc:TextArea;
private var mode_mc:Button;

/*** d) Embed new skins used by the Button component. ***/
// You can create a SWF file that contains symbols  with the names
// ModalUpSkin, ModalOverSkin, and ModalDownSkin.
// If you do not have skins, comment out these lines.
[Embed(source="Modal2.swf", symbol="blueCircle")]
public var modeUpSkinName:Class;

[Embed(source="Modal2.swf", symbol="blueCircle")]
public var modeOverSkinName:Class;
[Embed(source="Modal2.swf", symbol="greenSquare")]
public var modeDownSkinName:Class;
/*** e) Implement the createChildren() method. ***/
// Test for the existence of the children before creating them.
// This is optional, but we do this so a subclass can create a
// different child instead.
override protected function createChildren():void {
    super.createChildren();
    // Create and initialize the TextArea control.
    if (!text_mc)
    {
        text_mc = new TextArea();
        text_mc.explicitWidth = 80;
        text_mc.editable = false;
        text_mc.text= _text;
        text_mc.addEventListener("change", handleChangeEvent);
        addChild(text_mc);
    }
    // Create and initialize the Button control.
    if (!mode_mc)
    {   mode_mc = new Button();
        mode_mc.label = "Toggle Editing Mode";
        // If you do not have skins available,
        // comment out these lines.
        mode_mc.setStyle('overSkin', modeOverSkinName);
        mode_mc.setStyle('upSkin', modeUpSkinName);
        mode_mc.setStyle('downSkin', modeDownSkinName);
        mode_mc.addEventListener("click", handleClickEvent);
        addChild(mode_mc);
    }
}

/*** f) Implement the commitProperties() method. ***/
override protected function commitProperties():void {
    super.commitProperties();

    if (bTextChanged) {
        bTextChanged = false;
        text_mc.text = _text;
        invalidateDisplayList();
    }
}
/*** g) Implement the measure() method. ***/
// The default width is the size of the text plus the button.
// The height is dictated by the button.
override protected function measure():void {
    super.measure();
    // Since the Button control uses skins, get the
    // measured size of the Button control.
    var buttonWidth:Number = mode_mc.getExplicitOrMeasuredWidth();
    var buttonHeight:Number = mode_mc.getExplicitOrMeasuredHeight();
    // The default and minimum width are the measuredWidth
    // of the TextArea control plus the measuredWidth
    // of the Button control.
    measuredWidth = measuredMinWidth =
        text_mc.measuredWidth + buttonWidth;

    // The default and minimum height are the larger of the
    // height of the TextArea control or the measuredHeight of the
    // Button control, plus a 10 pixel border around the text.
    measuredHeight = measuredMinHeight =
        Math.max(mode_mc.measuredHeight,buttonHeight) + 10;
}


/*** h) Implement the updateDisplayList() method. ***/
// Size the Button control to the size of its label text
// plus a 10 pixel border area.
// Size the TextArea to the remaining area of the component.
// Place the children depending on the setting of
// the textPlacement property.
override protected function updateDisplayList(unscaledWidth:Number,
        unscaledHeight:Number):void {
    super.updateDisplayList(unscaledWidth, unscaledHeight);
    // Subtract 1 pixel for the left and right border,
    // and use a 3 pixel margin on left and right.
    var usableWidth:Number = unscaledWidth - 8;
    // Subtract 1 pixel for the top and bottom border,
    // and use a 3 pixel margin on top and bottom.
    var usableHeight:Number = unscaledHeight - 8;

    // Calculate the size of the Button control based on its text.
    var lineMetrics:TextLineMetrics = measureText(mode_mc.label);
    // Add a 10 pixel border area around the text.
    var buttonWidth:Number = lineMetrics.width + 10;
    var buttonHeight:Number = lineMetrics.height + 10;
    mode_mc.setActualSize(buttonWidth, buttonHeight);

    // Calculate the size of the text
    // Allow for a 5 pixel gap between the Button
    // and the TextArea controls.
    var textWidth:Number = usableWidth - buttonWidth - 5;
    var textHeight:Number = usableHeight;
    text_mc.setActualSize(textWidth, textHeight);

    // Position the controls based on the textPlacement property.
    if (textPlacement == "left") {
        text_mc.move(4, 4);
        mode_mc.move(4 + textWidth + 5, 4);
    }
    else {
```

```
                mode_mc.move(4, 4);
                text_mc.move(4 + buttonWidth + 5, 4);
            }

            // Draw a simple border around the child components.
            graphics.lineStyle(1, 0x000000, 1.0);
            graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);
        }

        /*** i) Add methods, properties, and metadata. ***/
        // The general pattern for properties is to specify a private
        // holder variable.
        private var _textPlacement:String = "left";

        // Create a getter/setter pair for the textPlacement property.
        public function set textPlacement(p:String):void {
            _textPlacement = p;
            invalidateDisplayList();
            dispatchEvent(new Event("placementChanged"));
        }

        // The textPlacement property supports data binding.
        [Bindable(event="placementChanged")]
        public function get textPlacement():String {
            return _textPlacement;
        }

        private var _text:String = "ModalText";
        private var bTextChanged:Boolean = false;

        // Create a getter/setter pair for the text property.
        public function set text(t:String):void {
            _text = t;
            bTextChanged = true;
            invalidateProperties();
            dispatchEvent(new Event("textChanged"));
        }

        [Bindable(event="textChanged")]
        public function get text():String {
                return text_mc.text;
        }

        // Handle events that are dispatched by the children.
        private function handleChangeEvent(eventObj:Event):void {
                dispatchEvent(new Event("change"));
        }
        // Handle events that are dispatched by the children.
        private function handleClickEvent(eventObj:Event):void {
                text_mc.editable = !text_mc.editable;
        }
    }
}
```

Troubleshooting

I get an error “don't know how to parse…” when I try to use the component from MXML.

This means that the compiler could not find the SWC file, or the contents of the SWC file did not list the component. Ensure that the SWC file is in a directory that Flex searches, and ensure that your `xmlns` property is pointing to the right place. Try moving the SWC file to the same directory as the MXML file and setting the namespace to "*" as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns="*">
```

For more information, see “Flex compilers” on page 2164.

I get an error “xxx is not a valid attribute…” when I try to use the component from MXML.

Ensure that the attribute is spelled correctly. Also ensure that it is not private.

I don't get any errors, but nothing appears.

Verify that the component was instantiated. One way to do this is to put a Button control and a TextArea control in the MXML application and set the `text` property to the ID for the component when the button is clicked. For example:

```
<!-- This verifies whether a component was instantiated. -->
<zz:mycomponent id="foo"/>
<mx:TextArea id="output"/>
<mx:Button label="Print Output" click="output.text = foo.id;"/>
```

The component is instantiated properly but does not appear (1).

In some cases, helper classes are not ready by the time your component requires them. Flex adds classes to the application in the order that they must be initialized (base classes, and then child classes). However, if you have a static method that gets called as part of the initialization of a class, and that static method has class dependencies, Flex does not know to place that dependent class before the other class, because it does not know when that method is going to be called.

One possible remedy is to add a static variable dependency to the class definition. Flex knows that all static variable dependencies must be ready before the class is initialized, so it orders the class loading correctly.

The following example adds a static variable to tell the linker that class A must be initialized before class B:

```
public class A {

    static function foo():Number {
        return 5;
    }
}

public class B {
    static function bar():Number {
        return mx.example.A.foo();
    }

    static var z = B.bar();
    // Dependency
    static var ADependency:mx.example.A = mx.example.A;
}
```

The component is instantiated properly but does not appear (2).

Verify that the `measuredWidth` and `measuredHeight` properties are nonzero. If they are zero or `NaN`, ensure that you implemented the `measure()` method correctly.

You can also verify that the `visible` property is set to `true`. If `visible is false`, ensure that your component called the `invalidateDisplayList()` method.

The component is instantiated properly but does not appear (3).

It is possible that there is another class or SWC file that overrides your custom class or the symbols used in your component. Ensure that there are no naming conflicts.

Custom style properties

Styles are useful for defining the look and feel of your applications, including letting users set component skins. You can use them to change the appearance of a single component, or apply them across all components, including custom components.

About styles

You can modify the appearance of Flex components through style properties. These properties can define the size of a font used in a Label control, or the background color used in the Tree control. In Flex, some styles are inherited from parent containers to their children, and across style types and classes. This means that you can define a style once, and then have that style apply to all controls of a single type or to a set of controls. Also, you can override individual properties for each control at a local, document, or global level, giving you great flexibility in controlling the appearance of your applications.

For more information, see "Styles and themes" on page 1492.

About inheritance in Cascading Style Sheets

When you implement a style property in an ActionScript component, that property is automatically inherited by any subclasses of your class, just as methods and properties are inherited. This type of inheritance is called object-oriented inheritance.

Some style properties also support Cascading Style Sheet (CSS) inheritance. CSS inheritance means that if you set the value of a style property on a parent container, a child of that container inherits the value of the property when your application runs. For example, if you define the `fontFamily` style as Times for a Panel container, all children of that container use Times for `fontFamily`, unless they override that property.

In general, color and text styles support CSS inheritance, regardless of whether they are set by using CSS or style properties. All other styles do not support CSS inheritance, unless otherwise noted.

If you set a style on a parent that does not support CSS inheritance, such as `textDecoration`, only the parent container uses that value, and not its children. There is an exception to the rules of CSS inheritance. If you use the global type selector in a CSS style definition, Flex applies those style properties to all controls, regardless of whether the properties are inheritable.

For more information about style inheritance, see "About style inheritance" on page 1526.

About setting styles

Flex provide several ways of setting component styles: using MXML tag attributes, calling the setStyle() method, and CSS.

Setting styles using MXML tag attributes

Component users can use MXML tag attributes to set a style property on a component. For example, the following code creates a TextArea control, and then sets the `backgroundColor` style of the component to blue (0x0000FF):

```
<mx:TextArea id="myTA" backgroundColor="0x0000FF"/>
```

Setting styles using the setStyle() method

Component users can use the `setStyle()` method to set a style property on a component. For example, the following code creates a TextArea control, and then sets the `fontWeight` style of the component to bold:

```
var myTA:TextArea=new TextArea();
myTA.setStyle('fontWeight', 'bold');
```

Setting styles using CSS

Component users can use the `<fx:Styles>` tag to set CSS styles in an MXML application, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- createcomps_styles/CCStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*" >

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|TextArea {fontWeight: "bold"}
    </fx:Style>

    <s:TextArea/>

</s:Application>
```

You can also import an external CSS file, as the following example shows:

```
<fx:Style source="myStyle.css"/>
```

Overriding the styleChanged() method

When a user sets a style on a component, Flex calls the component's `styleChanged()` method, passing to it the name of the style being set. When you create a custom component, you can override the UIComponent.styleChanged() method to check the style name passed to it, and handle the change accordingly, as the following example shows:

```
var bBackgroundColor:Boolean=false;

override public function styleChanged(styleProp:String):void {

    super.styleChanged(styleProp);

    // Check to see if style changed.
    if (styleProp=="backgroundColor")
    {
        bBackgroundColor=true;
        invalidateDisplayList();
        return;
    }
}
```

The `styleChanged()` method first calls superclass' `styleChanged()` method to let the superclass handle the style change.

After the superclass gets a call to handle the style change, your component can detect that the user set the `backgroundColor` style, and handle it. By handling the style change after the superclass makes the change, you can override the way the superclass handles the style.

Notice that the method calls the invalidateDisplayList() method, which causes Flex to execute the component's `updateDisplayList()` method at the next screen update. Although you can detect style changes in the `styleChanged()` method, you still use the updateDisplayList() method to draw the component on the screen. For more information, see "Defining a style property" on page 2504.

Typically, you use a flag to indicate that a style changed. In the `updateDisplayList()` method, you check the flag and update the component based on the new style setting, as the following example shows:

```
override protected function updateDisplayList(unscaledWidth:Number, unscaledHeight:Number):void {

    super.updateDisplayList(unscaledWidth, unscaledHeight);

    // Check to see if style changed.
    if (bBackgroundColor==true)
    {
        // Redraw the component using the new style.
        ...
    }
}
```

By using flags to signal style updates to the `updateDisplayList()` method, the `updateDisplayList()` method has to perform only the updates based on the style changes; it may not have to redraw or recalculate the appearance of the entire component. For example, if you are changing only the border color of a component, it is more efficient to redraw only the border, rather than redrawing the entire component every time someone changes a style.

Example: Creating style properties

When you create a component, you might want to create a style property so that component users can configure it by using styles. For example, you create a component, named StyledRectangle, that uses a gradient fill pattern to define its color, as the following example shows:

This gradient is defined by two colors that you set by using a new style property called `fillColors`. The `fillColors` style takes an array of two colors that component users can set. The StyledRectangle.as class defines default colors for the `fillColors` style, but you can also set them, as the following example shows:

```
<?xml version="1.0"?>
<!-- skinstyle\MainRectWithFillStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Set style by using a CSS type selector. -->
    <fx:Style>
        @namespace "myComponents.*";
        StyledRectangle {fillColors: #FF00FF, #00FFFF}
    </fx:Style>
    <!-- By default, use the style defined by the CSS type selector. -->
    <MyComp:StyledRectangle id="mySR1"/>
    <!-- By default, use the style defined by the CSS type selector. -->
    <MyComp:StyledRectangle id="mySR2"/>
    <!-- Change the default style by using the setStyle() method. -->
    <mx:Button label="Set gradient"
        click="mySR2.setStyle('fillColors', [0x000000, 0xFFFFFF]);"/>
    <!-- Set fillColors in MXML. -->
    <MyComp:StyledRectangle id="mySR3" fillColors="[0x00FF00, 0xFFFFFF]"/>
</s:Application>
```

In this example, the CSS type selector for the StyledRectangle component sets the initial values of the `fillColors` property to `#FF00FF` and `#00FFFF`. For the second StyledRectangle components, you use the `click` event of a Button control to change the `fillColor` style by using the `setStyle()` method. The third component sets the style property by using an MXML tag attribute.

Defining a style property

You define a style property for a component in the class definition.

1. Insert the `[Style]` metadata tag that defines the style before the class definition.

   You insert the `[Style]` metadata tag before the class definition to define the MXML tag attribute for a style property. If you omit the `[Style]` metadata tag, the MXML compiler issues a syntax error when you try to set the property as an MXML tag attribute.

   The `[Style]` metadata tag has the following syntax:

   ```
   [Style(name="style_name"[,property="value",...])]
   ```

   For more information, see "Metadata tags in custom components" on page 2376.

2. Override the styleChanged() method to detect changes to the property.

3. Override updateDisplayList() method to incorporate the style into the component display.

4. Define a static initializer to set the default value of the style property.

   For more information, see "Setting default style values" on page 2506.

The following code example defines the StyledRectangle component and the `fillColors` style. It also defines a second style property named `alphas` that you can use to set the alpha values of the fill:

```
package myComponents
{
    // skinstyle/myComponents/StyledRectangle.as
    import mx.core.UIComponent;
    import mx.styles.CSSStyleDeclaration;
    import mx.styles.StyleManager;
    import flash.display.GradientType;
    import mx.core.FlexGlobals;
    // Insert the [Style] metadata tag to define the name, type
    // and other information about the style property for the
    // MXML compiler.
    [Style(name="fillColors",type="Array",format="Color",inherit="no")]
    [Style(name="alphas",type="Array",format="Number",inherit="no")]
    public class StyledRectangle extends UIComponent
    {
        // Define a static variable.
        private static var classConstructed:Boolean = classConstruct();

        // Define a static method.
        private static function classConstruct():Boolean {
            if 
(!FlexGlobals.topLevelApplication.styleManager.getStyleDeclaration("myComponents.StyledRecta
ngle"))
            {
                // If there is no CSS definition for StyledRectangle,
                // then create one and set the default value.
                var myRectStyles:CSSStyleDeclaration = new CSSStyleDeclaration();
                myRectStyles.defaultFactory = function():void
                {
                    this.fillColors = [0xFF0000, 0x0000FF];
                    this.alphas = [0.5, 0.5];
                }

FlexGlobals.topLevelApplication.styleManager.setStyleDeclaration("myComponents.StyledRectang
le", myRectStyles, true);
            }
            return true;
        }

        // Constructor
        public function StyledRectangle() {
            super();
        }

        // Define a default size of 100 x 100 pixels.
        override protected function measure():void {
            super.measure();
            measuredWidth = measuredMinWidth = 100;
            measuredHeight = measuredMinHeight = 100;
        }
        // Define the flag to indicate that a style property changed.
        private var bStypePropChanged:Boolean = true;
        // Define the variable to hold the current gradient fill colors.
        private var fillColorsData:Array;
        // Define the variable to hold the current alpha values.
        private var alphasData:Array;
        // Define the variable for additional control on the fill.
        // You can create a style property for this as well.
        private var ratios:Array = [0x00, 0xFF];

        // Override the styleChanged() method to detect changes in your new style.
        override public function styleChanged(styleProp:String):void {
            super.styleChanged(styleProp);
            // Check to see if style changed.
            if (styleProp=="fillColors" || styleProp=="alphas")
            {
                bStypePropChanged=true;
                invalidateDisplayList();
                return;
            }
        }


        // Override updateDisplayList() to update the component
        // based on the style setting.
        override protected function updateDisplayList(unscaledWidth:Number,
                unscaledHeight:Number):void {
            super.updateDisplayList(unscaledWidth, unscaledHeight);
            // Check to see if style changed.
            if (bStypePropChanged==true)
            {
                // Redraw gradient fill only if style changed.
                fillColorsData=getStyle("fillColors");
                alphasData=getStyle("alphas");

                graphics.beginGradientFill(GradientType.LINEAR,
                    fillColorsData, alphasData, ratios);
                graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);

                bStypePropChanged=false;
            }
        }
    }
}
```

Setting default style values

One of the issues that you have to decide when you create a style property for your component is how to set its default value. Setting a default value for a style property is not as simple as calling the setStyle`()` method in the component's constructor; you must take into consideration how Flex processes styles, and the order of precedence of styles.

When Flex compiles your application, Flex first examines any style definitions in the `<fx:Style>` tag, before it creates any components. Therefore, if you call `setStyle()` from within the component's constructor, which occurs after processing the `<fx:Style>` tag, you set the style property on each instance of the component; this overrides any conflicting CSS declarations in the `<fx:Style>` tag.

The easiest way to set a default value for a style property is to define a static initializer in your component. A static initializer is executed once, the first time Flex creates an instance of a component. In "Defining a style property" on page 2504, you defined a static initializer, by using the `classConstructed` variable and the `classConstruct()` method, as part of the StyledRectangle.as class.

The `classConstruct()` method is invoked the first time Flex creates a StyledRectangle component. This method determines whether a style definition for the StyledRectangle class already exists, defined by using the `<fx:Style>` tag. If no style is defined, the `classConstruct()` method creates one, and sets the default value for the style property.

Therefore, if you omit the `<fx:Style>` tag from your application, the style definition is created by the `classConstruct()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- skinstyle\MainRectNoStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <MyComp:StyledRectangle/>

</s:Application>
```

If you include the `<fx:Style>` tag, the `<fx:Style>` tag creates the default style definition, as the following example shows:

```
<?xml version="1.0"?>
<!-- skinstyle\MainRectCSSStyles.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*">

    <fx:Style>
        @namespace "myComponents.*";
        StyledRectangle {fillColors: #FF00FF, #00FFFF}
    </fx:Style>
    <MyComp:StyledRectangle/>

</s:Application>
```

Template components

One way to create reusable components is to define them as template components. A template component defines properties with a general data type that lets the component user specify an object of a concrete data type when using the component. By using a general data type to define component properties, you create highly reusable components that can work with many different types of objects.

About template components

A standard component defines a property with a concrete data type, such as Number or String. The component user must then pass a value that exactly matches the property's data type or else Adobe® Flex® issues a compiler error.

A *template component* is a component in which one or more of its properties is defined with a general data type. This property serves as a slot for values that can be of the exact data type of the property, or of a value of a subclass of the data type. For example, to accept any Flex visual component as a property value, you define the data type of the property as UIComponent. To accept only container components, you define the data type of the property as Container.

When you use the template component in an application, the component user sets the property value to be an object with a concrete data type. You can think of the property as a placeholder for information, where it is up to the component user, rather than the component developer, to define the actual data type of the property.

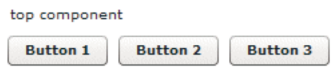
The following example shows an application that uses a template component called MyTemplateComponent:

```
<?xml version="1.0"?>
<!-- templating/MainTemplateButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*"
    height="700" width="700">
    <s:Panel>
        <MyComp:MyTemplateComponent id="myTComp1">
            <MyComp:topRow>
                <s:Label text="top component"/>
            </MyComp:topRow>
            <MyComp:bottomRow>
                <s:Button label="Button 1"/>
                <s:Button label="Button 2"/>
                <s:Button label="Button 3"/>
            </MyComp:bottomRow>
        </MyComp:MyTemplateComponent>
    </s:Panel>
</s:Application>
```

The MyTemplateComponent takes two properties:

- The `topRow` property specifies the single Flex component that appears in the top row of the VGroup container.
- The `bottomRow` property specifies one or more Flex components that appear along the bottom row of the VGroup container.

The implementation of the MyTemplateComponent consists of a VGroup container that displays its children in two rows. The following image shows the output of this application:



The implementation of the `topRow` and `bottomRow` properties lets you specify any Flex component as a value, as the following example shows:

```
<?xml version="1.0"?>
<!-- templating/MainTemplateLink.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*"
    height="700" width="700">
    <s:Panel>

        <MyComp:MyTemplateComponent id="myTComp2">
            <MyComp:topRow>
                <s:TextArea text="top component"/>
            </MyComp:topRow>
            <MyComp:bottomRow>
                <mx:LinkButton label="Link 1"/>
                <mx:LinkButton label="Link 2"/>
                <mx:LinkButton label="Link 3"/>
            </MyComp:bottomRow>
        </MyComp:MyTemplateComponent>
    </s:Panel>
</s:Application>
```

In this example, the top component is a TextArea control, and the bottom components are two LinkButton controls.

Implementing a template component

The section “About template components” on page 2507 shows an example of a template component named MyTemplateComponent. Flex provides you with two primary ways to create template components:

- Create properties with general data types, such as UIComponent or Container.
- Create properties with the type IDeferredInstance.

Using general data types in a template component

One way to implement the component MyTemplateComponent (see “About template components” on page 2507) is to define the properties `topRow` and `bottomRow` as type UIComponent. Users of the component can specify any object to these properties that is an instance of the UIComponent class, or an instance of a subclass of UIComponent.

The following code shows the implementation of MyTemplateComponent:

```
<?xml version="1.0"?>
<!-- templating/myComponents/MyTemplateComponent.mxml -->
<s:VGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="init();">
    <fx:Script>
        <![CDATA[

            import spark.components.HGroup;
            import mx.core.UIComponent;

            // Define a property for the top component.
            public var topRow:UIComponent;
            // Define an Array of properties for a row of components.
            // Restrict the type of the Array elements
            // to mx.core.UIComponent.
            [ArrayElementType("mx.core.UIComponent")]
            public var bottomRow:Array;

            private function init():void {
                // Add the top component to the VGroup container.
                addElement(topRow);
                // Create an HGroup container. This container
                // is the parent container of the bottom row of components.
                var controlHGroup:HGroup = new HGroup();
                // Add the bottom row of components
                // to the HGroup container.
                for (var i:int = 0; i < bottomRow.length; i++)
                    controlHGroup.addElement(bottomRow[i]);
                // Add the HGroup container to the VGroup container.
                addElement(controlHGroup);
            }
        ]]>
    </fx:Script>
</s:VGroup>
```

For the `bottomRow` property, you define it as an Array and include the `[ArrayElementType]` metadata tag to specify to the compiler that the data type of the Array elements is also UIComponent. For more information on the `[ArrayElementType]` metadata tag, see “Metadata tags in custom components” on page 2376.

Using IDeferredInstance in a template component

Deferred creation is a feature of Flex where Flex containers create only the controls that initially appear to the user. Flex then creates the container’s other descendants if the user navigates to them. For more information, see “Improving startup performance” on page 2333.

You can create a template component that also takes advantage of deferred creation. Rather than having Flex create your component and its properties when the application loads, you can define a component that creates its properties only when a user navigates to the area of the application that uses the component. This is especially useful for large components that may have many child components. Flex view states make use of this feature.

The following example shows an alternative implementation for the MyTemplateComponent component shown in the section “About template components” on page 2507, named MyTemplateComponentDeferred.mxml, by defining the topRow and bottomRow properties to be of type IDeferredInstance:

```
<?xml version="1.0"?>
<!-- templating/myComponents/MyTemplateComponentDeferred.mxml -->
<s:VGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="init();">
    <fx:Script>
        <![CDATA[

            import spark.components.HGroup;
            import mx.core.UIComponent;

            // Define a deferred property for the top component.
            public var topRow:IDeferredInstance;
            // Define an Array of deferred properties
            // for a row of components.
            [ArrayElementType("mx.core.IDeferredInstance")]
            public var bottomRow:Array;

            private function init():void {
                // Add the top component to the VGroup container.
                // Cast the IDeferredInstance object to UIComponent
                // so that you can add it to the parent container.
                addElement(UIComponent(topRow.getInstance()));
                // Create an HGroup container. This container
                // is the parent container of the bottom row of components.
                var controlHGroup:HGroup = new HGroup();

                // Add the bottom row of components
                // to the HGroup container.
                for (var i:int = 0; i < bottomRow.length; i++)
            controlHGroup.addElement(UIComponent(bottomRow[i].getInstance()));
                // Add the HBox container to the VGroup container.
                addElement(controlHGroup);
            }
        ]]>
    </fx:Script>
</s:VGroup>
```

The IDeferredInstance interface defines a single method, `getInstance()`. Flex calls the `getInstance()` method to initialize a property when it creates an instance of the component. A subsequent call to the `getInstance()` method returns a reference to the property value.

In MXML, when the compiler encounters a value declaration for a property of type IDeferredInstance, instead of generating code to construct and assign the value to the property, the compiler generates code to construct and assign an IDeferredInstance implementation object, which then produces the value at run time.

You can pass any data type to a property of type IDeferredInstance. In the example in the section “About template components” on page 2507, you pass a Label control to the `topRow` property, and three Button controls to the `bottomRow` property.

Notice in the example that the `addElement()` methods that take `topRow` and `bottomRow` as arguments cast them to UIComponent. This cast is necessary because the `addElement()` method can only add an object that implements the IUIComponent interface to a container, and the `DeferredInstance.getInstance()` method returns a value of type Object.

Defining properties using the IDeferredInstance interface

You can define component properties of type IDeferredInstance.

Defining a generic property

To define a generic property, one with no associated data type, you define its type as IDeferredInstance, as the following example shows:

```
// Define a deferred property for the top component.
public var topRow:IDeferredInstance;
```

The user of the component can then specify an object of any type to the property. It is your responsibility in the component implementation to verify that the value passed by the user is of the correct data type.

Restricting the data type of a property

You use the `[InstanceType]` metadata tag to specify the allowed data type of a property of type IDeferredInstance, as the following example shows:

```
// Define a deferred property for the top component.
[InstanceType("spark.components.Label")]
public var topRow:IDeferredInstance;
```

The Flex compiler validates that users only assign values of the specified type to the property. In this example, if the component user sets the `topRow` property to a value of a type other than spark.components.Label, the compiler issues an error message.

Defining an array of template properties

You can define an Array of template properties, as the following example shows:

```
// Define an Array of deferred properties for a row of components.
// Do not restrict the type of the component.
[ArrayElementType("mx.core.IDeferredInstance")]
public var bottomRow:Array;

// Define an Array of deferred properties for a row of components.
// Restrict the type of the component to mx.controls.Button.
[InstanceType("mx.controls.Button")]
[ArrayElementType("mx.core.IDeferredInstance")]
public var bottomRow:Array;
```

In the first example, you can assign a value of any data type to the `bottomRow` property. Each array element's `getInstance()` method is not called until the element is used.

In the second example, you can only assign values of type spark.components.Button to it. Each Array element is created when the application loads. The following template component shows an alternative implementation of the MyTemplateComponent that restricts the type of components to be of type mx.controls.Button:

```
<?xml version="1.0"?>
<!-- templating/myComponents/MyTemplateComponentDeferredSpecific.mxml -->
<s:VGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    initialize="init();">

    <fx:Script>
        <![CDATA[

            import spark.components.HGroup;
            import mx.core.UIComponent;

            [InstanceType("spark.components.Label")]
            public var topRow:IDeferredInstance;
            // Define an Array of deferred properties
            // for a row of components.
            // Restrict the type of the component
            // to mx.controls.Button.
            [InstanceType("spark.components.Button")]
            [ArrayElementType("mx.core.IDeferredInstance")]
            public var bottomRow:Array;

            private function init():void {
                addElement(UIComponent(topRow.getInstance()));
                var controlHGroup:HGroup = new HGroup();
                for (var i:int = 0; i < bottomRow.length; i++)
            controlHGroup.addElement(UIComponent(bottomRow[i].getInstance()));
                addElement(controlHGroup);
            }
        ]]>
    </fx:Script>
</s:VGroup>
```

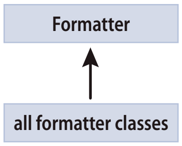
Custom formatters

Adobe® Flex® includes several predefined formatters that you can use in your applications to format data. You also might have to extend the functionality of these predefined formatters, or create formatters for your specific application needs.

For more information on using formatters, see “Formatting Data” on page 2004.

Creating a custom formatter

You create a custom formatter by creating a class that extends the mx.formatters.Formatter base class, or by creating a class that extends one of the standard formatter classes, which all extend mx.formatters.Formatter. The following example shows the class hierarchy for formatters:



Like standard formatter classes, your custom formatter class must contain a public `format()` method that takes a single argument and returns a String that contains the formatted data. Most of the processing of your custom formatter occurs within the `format()` method.

Your custom formatter also might let the user specify which pattern formats the data. Where applicable, the Flex formatters, such as the ZipCodeFormatter, use a `formatString` property to pass a format pattern. Some Flex formatters, such as the NumberFormatter and CurrencyFormatter classes, do not have `formatString` properties, because they use a set of properties to configure formatting.

Creating a simple formatter

This example defines a simple formatter class that converts any String to all uppercase or all lowercase letters depending on the value passed to the `formatString` property. By default, the formatter converts a String to all uppercase.

```
package myFormatters
{
    // formatters/myFormatter/SimpleFormatter.as
    import mx.formatters.Formatter
    import mx.formatters.SwitchSymbolFormatter
    public class SimpleFormatter extends Formatter
    {
        // Declare the variable to hold the pattern string.
        public var myFormatString:String = "upper";
        // Constructor
        public function SimpleFormatter() {
            // Call base class constructor.
            super();
        }
        // Override format().
        override public function format(value:Object):String {
            // 1. Validate value - must be a nonzero length string.
            if( value.length == 0)
                {   error="0 Length String";
                    return ""
                }
            // 2. If the value is valid, format the string.
            switch (myFormatString) {
                case "upper" :
                    var upperString:String = value.toUpperCase();
                    return upperString;
                    break;
                case "lower" :
                    var lowerString:String = value.toLowerCase();
                    return lowerString;
                    break;
                default :
                    error="Invalid Format String";
                    return ""
            }
        }
    }
}
```

You can use this formatter in an application, as the following example shows:

```
<?xml version="1.0" ?>
<!-- createcomps_formatters/FormatterSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myFormatters.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Declare a formatter and specify formatting properties. -->
        <MyComp:SimpleFormatter id="upperFormat" myFormatString="upper" />
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput id="myTI" />

    <mx:TextArea text="Your uppercase string is {upperFormat.format(myTI.text)}" />
</s:Application>
```

The namespace declaration in the `<s:Application>` tag specifies to use the `MyComp` prefix when referencing the formatter, and the location of the formatter's ActionScript file. That file is in the myFormatters subdirectory of the application, or in the default classpath of the application. For more information on deploying your formatters, see "Component compilation" on page 2392.

Handling errors in formatters

For all formatter classes, except for the SwitchSymbolFormatter class, when an error occurs, the formatter returns an empty string and writes a string that describes the error condition to the formatter's `error` property. The `error` property is inherited from the Formatter superclass.

In your application, you can test for an empty string in the result returned by the formatter. If detected, you can check the `error` property to determine the cause of the error. For an example that handles a formatter error, see "Formatting Data" on page 2004. For more information on the SwitchSymbolFormatter class, see "Using the SwitchSymbolFormatter class" on page 2515.

Using the SwitchSymbolFormatter class

You can use the SwitchSymbolFormatter utility class when you create custom formatters. You use this class to replace placeholder characters in one string with numbers from a second string.

For example, you specify the following information to the SwitchSymbolFormatter class:

Format string The Social Security number is: ###-##-####"

Input string "123456789"

The SwitchSymbolFormatter class parses the format string and replaces each placeholder character with a number from the input string in the order in which the numbers are specified in the input string. The default placeholder character is the number sign (#). You can define a different placeholder character by passing it to the constructor when you create a SwitchSymbolFormatter object. For an example, see "Using a different placeholder character" on page 2516.

The SwitchSymbolFormatter class creates the following output string from the Format and Input strings:

```
"The Social Security number is: 123-45-6789"
```

You pass the format string and input string to the SwitchSymbolFormatter.formatValue() method to create the output string, as the following example shows:

```
<?xml version="1.0" ?>
<!-- createcomps_formatters/FormatterSwitchSymbol.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
  <fx:Script>
    <![CDATA[

      import mx.formatters.SwitchSymbolFormatter;
      // Event handler to validate and format input.
      private function formatVal():void {

        var switcher:SwitchSymbolFormatter=new SwitchSymbolFormatter('#');
        formattedSCNumber.text =
          switcher.formatValue("Formatted Social Securty number: ###-##-#### ", scNum.text);
      }
    ]]>
  </fx:Script>
    <mx:Label
        text="Enter a 9 digit Social Security number with no separator characters:"/>
    <mx:TextInput id="scNum"
        text=""
        maxChars="9" width="50%"/>
  <mx:Button label="Format"
      click="formatVal();"/>
  <mx:TextInput id="formattedSCNumber"
      editable="false" width="75%"/>
</s:Application>
```

You can mix alphanumeric characters and placeholder characters in this format string. The format string can contain any characters that are constant for all values of the numeric portion of the string. However, the input string for formatting must be numeric. The number of digits supplied in the source value must match the number of digits defined in the format string.

Using a different placeholder character

By default, the SwitchSymbolFormatter class uses a number sign (#) as the placeholder character to indicate a number substitution within its format string. However, sometimes you might want to include a number sign in your actual format string. Then, you must use a different symbol to indicate a number substitution slot within the format string. You can select any character for this alternative symbol as long as it doesn’t appear in the format string.

For example, to use the ampersand character (&) as the placeholder, you create an instance of the SwitchSymbolFormatter class, as the following example shows:

```
var dataFormatter = new SwitchSymbolFormatter("&");
```

Handling errors with the SwitchSymbolFormatter class

Unlike other formatters, the SwitchSymbolFormatter class does not write its error messages into an `error` property. Instead, it is your responsibility to test for error conditions and return an error message if appropriate.

The custom formatter component in the following example formats nine-digit Social Security numbers by using the SwitchSymbolFormatter class:

```
package myFormatters
{
    // formatters/myFormatter/CustomSSFormatter.as
    import mx.formatters.Formatter
    import mx.formatters.SwitchSymbolFormatter
    public class CustomSSFormatter extends Formatter
    {
        // Declare the variable to hold the pattern string.
        public var formatString : String = "###-##-####";
        // Constructor
        public function CustomSSFormatter() {
            // Call base class constructor.
            super();
        }
        // Override format().
        override public function format( value:Object ):String {
            // Validate input string value - must be a 9-digit number.
            // You must explicitly check if the value is a number.
            // The formatter does not do that for you.
            if( !value  || value.toString().length != 9)
                {   error="Invalid String Length";
                    return ""
                }
            // Validate format string.
            // It must contain 9 number placeholders.
            var numCharCnt:int = 0;
            for( var i:int = 0; i<formatString.length; i++ )
                {
                    if( formatString.charAt(i) == "#" )
                    {   numCharCnt++;
                    }
                }
            if( numCharCnt != 9 )
            {
                error="Invalid Format String";
                return ""
            }
            // If the formatString and value are valid, format the number.
            var dataFormatter:SwitchSymbolFormatter =
                new SwitchSymbolFormatter();
            return dataFormatter.formatValue( formatString, value );
        }
    }
}
```

The following example uses this custom formatter in an application:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- createcomps_formatters/FormatterSS.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myFormatters.*">
    <fx:Declarations>
        <!-- Declare a formatter and specify formatting properties. -->
        <MyComp:CustomSSFormatter id="SSFormat"
            formatString="SS: #-#-#-#-#-#-#-#-#"/>
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput text="Your SS number is {SSFormat.format('123456789')}"/>
</s:Application>
```

Extending a Formatter class

You can extend the Formatter class to create a custom formatter, or any formatter class. The example in this section extends the ZipCodeFormatter class by allowing an extra format pattern: "#####*####".

In this example, if the user omits a format string, or specifies the default value of "#####*####", the formatter returns the ZIP code using the format "#####*####". If the user specifies any other format string, such as a five-digit string in the form "#####", the custom formatter calls the `format()` method in the superclass ZipCodeFormatter class to format the data.

```
package myFormatters
{
    // formatters/myFormatter/ExtendedZipCodeFormatter.as
    import mx.formatters.Formatter
    import mx.formatters.ZipCodeFormatter
    import mx.formatters.SwitchSymbolFormatter
    public class ExtendedZipCodeFormatter extends ZipCodeFormatter {
        // Constructor
        public function ExtendedZipCodeFormatter() {
            // Call base class constructor.
            super();
            // Initialize formatString.
            formatString = "#####*####";
        }
        // Override format().
        override public function format(value:Object):String {
            // 1. If the formatString is our new pattern,
            // then validate and format it.
            if( formatString == "#####*####" ){
                if( String( value ).length == 5 )
                    value = String( value ).concat("0000");
                if( String( value ).length == 9 ){
                    var dataFormatter:SwitchSymbolFormatter =
                        new SwitchSymbolFormatter();
                    return dataFormatter.formatValue( formatString, value );
                }
                else {
                    error="Invalid String Length";
                    return ""
                    }
                }
            // If the formatString is anything other than '#####*####,
            // call super and validate and format as usual using
            // the base ZipCodeFormatter.
            return super.format(value);
        }
    }
}
```

Notice that the ExtendedZipCodeFormatter class did not have to define a `formatString` property because it is already defined in its base class, ZipCodeFormatter.

The following example uses this custom formatter in an application:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- createcomps_formatters/FormatterZC.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myFormatters.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Declare a formatter and specify formatting properties. -->
        <MyComp:ExtendedZipCodeFormatter id="ZipCodeFormat"/>
    </fx:Declarations>
    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput width="220"
        text="Your zipcode number is {ZipCodeFormat.format('123456789')}"/>
</s:Application>
```

Custom validators

Data validators let you validate the data in an object. Adobe® Flex® supplies a number of standard validators that you can use in your application, but you can also define custom validators for your specific application needs.

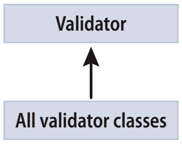
For information on the standard validators, see “Validating Data” on page 1964.

Validating data by using custom validators

The data that a user enters in a user interface might or might not be appropriate for the application. In Flex, you use a *validator* to ensure the values in the fields of an object meet certain criteria. For example, you can use a validator to ensure that a user enters a valid phone number value in a TextInput control.

Flex includes a set of validators for common types of user input data, such as ZIP codes, phone numbers, and credit cards. Although Flex supplies a number of commonly used validators, your application may require you to create custom validator classes. The mx.validators.Validator class is an ActionScript class that you can extend to add your own validation logic. Your classes can extend the functionality of an existing validator class, or you can implement new functionality in your custom validator class.

The following image shows the class hierarchy for validators:



About overriding the doValidation() method

Your custom validator class must contain an override of the protected Validator.doValidation() method that takes a single argument, `value`, of type Object, and returns an Array of ValidationResult objects. You return one ValidationResult object for each field that the validator examines and that fails the validation. For fields that pass the validation, you omit the ValidationResult object.

You do not have to create a ValidationResult object for fields that validate successfully. Flex creates those ValidationResult objects for you.

The base Validator class implements the logic to handle required fields by using the `required` property. When set to `true`, this property specifies that a missing or empty value in a user-interface control causes a validation error. To disable this verification, set this property to `false`.

In the `doValidation()` method of your validator class, you typically call the base class's `doValidation()` method to perform the verification for a required field. If the user did not enter a value, the base class issues a validation error stating that the field is required.

The remainder of the `doValidation()` method contains your custom validation logic.

About the ValidationResult class

The `doValidation()` method returns an Array of ValidationResult objects, one for each field that generates a validation error. The ValidationResult class defines several properties that let you record information about any validation failures, including the following:

errorCode A String that contains an error code. You can define your own error codes for your custom validators.

errorMessage A String that contains the error message. You can define your own error messages for your custom validators.

isError A Boolean value that indicates whether or not the result is an error. Set this property to `true`.

subField A String that specifies the name of the subfield associated with the ValidationResult object.

In your override of the `doValidation()` method, you can define an empty Array and populate it with ValidationResult objects as your validator encounters errors.

About the validate() method

You use the `Validator.validate()` method to programmatically invoke a validator from within a Flex application. However, you should never override this method in your custom validator classes. You need to override only the `doValidation()` method.

Example: Creating a simple validator

You can use the StringValidator class to validate that a string is longer than a minimum length and shorter than a maximum length, but you cannot use it to validate the contents of a string. This example creates a simple validator class that determines if a person is more than 18 years old based on their year of birth.

This validator extends the Validator base class, as the following example shows:

```
package myValidators
{
    import mx.validators.Validator;
    import mx.validators.ValidationResult;
    public class AgeValidator extends Validator {
        // Define Array for the return value of doValidation().
        private var results:Array;
        // Constructor.
        public function AgeValidator() {
            // Call base class constructor.
            super();
        }

        // Define the doValidation() method.
        override protected function doValidation(value:Object):Array {

            // Convert value to a Number.
            var inputValue:Number = Number(value);
            // Clear results Array.
            results = [];
            // Call base class doValidation().
            results = super.doValidation(value);
            // Return if there are errors.
            if (results.length > 0)
                return results;

            // Create a variable and initialize it to the current date.
            var currentYear:Date = new Date();

            // If input value is not a number, or contains no value,
            // issue a validation error.
            if (isNaN(inputValue) || !value )
            {
                results.push(new ValidationResult(true, null, "NaN",
                    "You must enter a year."));
                return results;
            }
            // If calculated age is less than 18, issue a validation error.
            if ((currentYear.getFullYear() - inputValue) < 18) {
                results.push(new ValidationResult(true, null, "tooYoung",
                    "You must be 18."));
                return results;
            }

            return results;
        }
    }
}
```

This example first defines a public constructor that calls `super()` to invoke the constructor of its base class. The base class can perform the check to ensure that data was entered into a required field, if you set the `required` property of the validator to `true`.

Notice that the second argument of the constructor for the ValidationResult class is `null`. You use this argument to specify a subfield, if any, of the object being validated that caused the error. When you are validating a single field, you can omit this argument. For an example that validates multiple fields, see “Example: Validating multiple fields” on page 2523.

You can use this validator in your Flex application, as the following example shows:

```
<?xml version="1.0" ?>
<!-- createcomps_validators/MainAgeValidator.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myValidators.*">
    <fx:Declarations>
        <MyComp:AgeValidator id="ageV"
            required="true"
            source="{birthYear}"
            property="text" />
    </fx:Declarations>

    <s:Form >
        <s:FormItem label="Enter birth year: ">
            <s:TextInput id="birthYear"/>
        </s:FormItem>
        <s:FormItem label="Enter birth year: ">
            <s:Button label="Submit"/>
        </s:FormItem>
    </s:Form>
</s:Application>
```

The `package` statement for your custom validator specifies that you should deploy it in a directory called `myValidators`. In the previous example, you place it in the subdirectory of the directory that contains your Flex application. Therefore, the namespace definition in your Flex application is `xmlns:MyComp="myValidators.*"`. For more information on deployment, see “Component compilation” on page 2392.

Example: Validating multiple fields

A validator can validate more than one field at a time. For example, you could create a custom validator called NameValidator to validate three input controls that represent a person’s first, middle, and last names.

To create a validator that examines multiple fields, you can either define properties on the validator that let you specify the multiple input fields, as does the Flex DateValidator class, or you can require that the single item passed to the validator includes all of the fields to be validated.

In the following example, you use a NameValidator that validates an item that contains three fields named `first`, `middle`, and `last`:

```
<?xml version="1.0" ?>
<!-- createcomps_validators/MainNameValidator.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myValidators.*">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:Model id="person">
            <name>
              <custName>
                 <first>{firstInput.text}</first>
                 <middle>{middleInput.text}</middle>
                 <last>{lastInput.text}</last>
              </custName>
            </name>
        </fx:Model>
        <MyComp:NameValidator id="nameVal"
            source="{person}" property="custName"
            listener="{firstInput}"/>
    </fx:Declarations>
    <mx:TextInput id="firstInput"/>
    <mx:TextInput id="middleInput"/>
    <mx:TextInput id="lastInput"/>
    <mx:Button label="Validate"
        click="nameVal.validate();"/>
</s:Application>
```

This validator examines three input fields. You specify `firstInput` as the validation listener. Therefore, when a validation error occurs, Flex shows a validation error message on the first TextInput control.

You can implement the NameValidator class, as the following example shows:

```
package myValidators
{
    import mx.validators.Validator;
    import mx.validators.ValidationResult;
    public class NameValidator extends Validator {
        // Define Array for the return value of doValidation().
        private var results:Array;
        public function NameValidator () {
            super();
        }
        override protected function doValidation(value:Object):Array {

            var fName:String = value.first;
            var mName:String = value.middle;
            var lName:String = value.last;

            // Clear results Array.
            results = [];
            // Call base class doValidation().
            results = super.doValidation(value);
            // Return if there are errors.
            if (results.length > 0)
```

```
                return results;
            // Check first name field.
            if (fName == "" || fName == null) {
                results.push(new ValidationResult(true,
                    "first", "noFirstName", "No First Name."));
                return results;
            }

            // Check middle name field.
            if (mName == "" || mName == null) {
                results.push(new ValidationResult(true,
                    "middle", "noMiddleName", "No Middle Name."));
                return results;
            }
            // Check last name field.
            if (lName == "" || lName == null) {
                results.push(new ValidationResult(true,
                    "last", "noLastName", "No Last Name."));
                return results;
            }

            return results;
        }
    }
}
```

In this example, because you are using a single validator to validate three subfields of the Object passed to the validator, you include the optional second argument to the constructor for the ValidationResult class to specify the subfield that caused the validation error. This inclusion permits Flex to identify the input component that caused the error, and to highlight that component in the application.

Custom effects

Adobe® Flex® supplies a number of standard effects that you can use in your application. However, you also can define custom effects for your specific application needs.

For information on the standard effects, see “Introduction to effects” on page 1784.

About creating a custom effect

Flex implements effects by using an architecture in which each effect is represented by two classes: a factory class and an instance class. Therefore, to implement a custom effect, you create two classes: the factory class and the instance class.

You create a factory class by creating a subclass of the mx.effects.Effect class, or by creating a subclass of one of the subclasses of the mx.effects.Effect class. You create an instance class by creating a subclass of the mx.effects.EffectInstance class, or a subclass of one of the subclasses of the mx.effects.EffectInstance class.

Defining factory and instance classes

To define a custom effect, you create two classes: the factory class and the instance class:

Factory class The factory class creates an object of the instance class to perform the effect on the target. You create a factory class instance in your application, and configure it with the necessary properties to control the effect, such as the zoom size or effect duration, as the following example shows:

```
<fx:Declarations>
    <!-- Define factory class. -->
    <s:Resize id="myResizeEffect" target="{myImage}"
        widthBy="10" heightBy="10"/>
</fx:Declarations>

<mx:Image id="myImage" source="@Embed(source='assets/logo.jpg')"/>
<s:Button label="Resize Me" click="myResizeEffect.end();myResizeEffect.play();"/>
```

By convention, the name of a factory class is the name of the effect, such as Zoom or Fade.

Instance class The instance class implements the effect logic. When you invoke an effect, the factory class creates an object of the instance class to perform the effect on the target. When the effect ends, Flex destroys the instance object. If the effect has multiple target components, the factory class creates multiple instance objects, one per target.

By convention, the name of an instance class is the name of the effect with the suffix *Instance*, such as ZoomInstance or FadeInstance.

About the effect base classes

You define effects by creating a subclass from the effects class hierarchy. Typically, you create a subclass from one of the following classes:

- mx.effects.Effect Create a subclass from this class for simple effects that do not require an effect to play over a period of time. For example, the Pause effect inserts a delay between two consecutive effects. You can also define a simple sound effect that plays an MP3 file.
- spark.effects.Animate Create a subclass from this class to define an effect that animates a set of properties between values over a period of time. For example, the Spark AnimateColor effect is a subclass of the Animate class that modifies a color property of its target over a specified duration.
- mx.effects.TweenEffect Create a subclass from this class to define an effect that plays over a period of time, such as an animation. For example, the MX Resize effect is a subclass of the TweenEffect class that modifies the size of its target over a specified duration.

About implementing your effects classes

You must override several methods and properties in your custom effect classes, and define any new properties and methods that are required to implement the effect. You can optionally override additional properties and methods based on the type of effect that you create.

The following table lists the methods and properties that you define in a factory class:

Factory method/property	**Description**
`constructor`	(Required) The class constructor. You typically call the `super()` method to invoke the superclass constructor to initialize the inherited items from the superclasses. Your constructor must take at least one optional argument, of type Object. This argument specifies the target component of the effect.
`Effect.initInstance()`	(Required) Copies properties of the factory class to the instance class. Flex calls this protected method from the `Effect.createInstance()` method; you do not have to call it yourself. In your override, you must call the `super.initInstance()` method.
`Effect.getAffectedProperties()`	(Required) Returns an Array of Strings, where each String is the name of a property of the target object that is changed by this effect. If the effect does not modify any properties, it should return an empty Array.
`Effect.instanceClass`	(Required) Contains an object of type Class that specifies the name of the instance class for this effect class. All subclasses of the Effect class must set this property, typically in the constructor.
`Effect.effectEndHandler()`	(Optional) Called when an effect instance finishes playing. If you override this method, ensure that you call the `super()` method.
`Effect.effectStartHandler()`	(Optional) Called when the effect instance starts playing. If you override this method, ensure that you call the `super()` method.
Additional methods and properties	(Optional) Define any additional methods and properties that the user requires to configure the effect.

The following table lists the methods and properties that you define in an instance class:

Instance method/property	**Description**
constructor	(Required) The class constructor. You typically call the `super()` method to invoke the superclass constructor to initialize the inherited items from the superclasses.
`EffectInstance.play()`	(Required) Invokes the effect. You must call `super.play()` from your override.
`EffectInstance.end()`	(Optional) Interrupts an effect that is currently playing, and jumps immediately to the end of the effect.
`EffectInstance.initEffect()`	(Optional) Called if the effect was triggered by the EffectManager. You rarely have to implement this method. For more information, see "Overriding the initEffect() method" on page 2542.
`TweenEffectInstance.onTweenUpdate()`	(Required) Use when you create a subclass from TweenEffectInstance. A callback method called at regular intervals to implement a tween effect. For more information, see "Example: Creating a tween effect" on page 2532.
`TweenEffectInstance.onTweenEnd()`	(Optional) Use when you create a subclass from TweenEffectInstance. A callback method called when the tween effect ends. You must call `super.onTweenEnd()` from your override. For more information, see "Example: Creating a tween effect" on page 2532.
Additional methods and properties	(Optional) Define any additional methods and properties. These typically correspond to the public properties and methods from the factory class, and any additional properties and methods that you require to implement the effect.

Example: Defining a simple effect

To define a simple custom effect, you create a factory class from the Effect base class, and the instance class from the mx.effects.EffectInstance class. The following example shows an effect class that uses a Sound object to play an embedded MP3 file when a user action occurs. This example is a simplified version of the SoundEffect class that ships with Flex.

```
package myEffects
{
   // createcomps_effects/myEffects/MySound.as
   import mx.effects.Effect;
   import mx.effects.EffectInstance;
   import mx.effects.IEffectInstance;
   public class MySound extends Effect
   {
       // Define constructor with optional argument.
       public function MySound(targetObj:Object = null) {
           // Call base class constructor.
           super(targetObj);

           // Set instanceClass to the name of the effect instance class.
           instanceClass= MySoundInstance;
       }

       // This effect modifies no properties, so your
       // override of getAffectedProperties() method
       // returns an empty array.
       override public function getAffectedProperties():Array {
           return [];
       }

       // Override initInstance() method.
       override protected function initInstance(inst:IEffectInstance):void {
           super.initInstance(inst);
       }
   }
}
```

The `package` statement in your class specifies that you should deploy it in a directory called myEffects. In this example, you place it in the subdirectory of the directory that contains your Flex application. Therefore, the namespace definition in your Flex application is `xmlns:MyComp="myEffects.*"`. For more information on deployment, see "Component compilation" on page 2392.

To define your instance class, you create a subclass from the mx.effects.EffectInstance class. In the class definition, you must define a constructor and `play()` methods, and you can optionally define an `end()` method to stop the effect.

```
package myEffects
{
    // createcomps_effects/myEffects/MySoundInstance.as
    import mx.effects.EffectInstance;
    import flash.media.SoundChannel;
    import flash.media.Sound;
    public class MySoundInstance extends EffectInstance
    {
        // Embed the MP3 file.
        [Embed(source="sample.mp3")]
        [Bindable]
        private var sndCls:Class;

        // Define local variables.
        private var snd:Sound = new sndCls() as Sound;
        private var sndChannel:SoundChannel;

        // Define constructor.
        public function MySoundInstance(targetObj:Object) {
            super(targetObj);
        }
        // Override play() method.
        // Notice that the MP3 file is embedded in the class.
        override public function play():void {
            super.play();
            sndChannel=snd.play();
        }

        // Override end() method class to stop the MP3.
        override public function end():void {
            sndChannel.stop();
            super.end();
        }
    }
}
```

To use your custom effect class in an MXML file, you insert a tag with the same name as the factory class in the MXML file. You reference the custom effect the same way that you reference a standard effect.

The following example shows an application that uses the MySound effect:

```
<?xml version="1.0"?>
<!-- createcomps_effects/MainSoundEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myEffects.*">
    <fx:Declarations>
        <MyComp:MySound id="mySoundEffect" target="{myLabel}" />
    </fx:Declarations>
    <!-- Use the SoundEffect effect with a mouseOver trigger. -->
    <s:Label id="myLabel"
        text="play MP3"
        rollOver="mySoundEffect.end();mySoundEffect.play();"/>
</s:Application>
```

Example: Passing parameters to effects

To make your effects more robust, you often design them to let the user pass parameters to them. The example in this section modifies the sound effect from the previous section to take a parameter that specifies the MP3 file to play:

```
package myEffects
{
    // createcomps_effects/myEffects/MySoundParam.as
    import mx.effects.Effect;
    import mx.effects.EffectInstance;
    import mx.effects.IEffectInstance;
    public class MySoundParam extends Effect
    {

       // Define a variable for the MP3 URL
       // and give it a default value.
       public var soundMP3:String=
           "http://localhost:8100/flex/assets/default.mp3";

       // Define constructor with optional argument.
       public function MySoundParam(targetObj:Object = null) {
           // Call base class constructor.
           super(targetObj);
           // Set instanceClass to the name of the effect instance class.
          instanceClass= MySoundParamInstance;
       }
       // Override getAffectedProperties() method to return an empty array.
       override public function getAffectedProperties():Array {
           return [];
       }

       // Override initInstance() method.
       override protected function initInstance(inst:IEffectInstance):void {
           super.initInstance(inst);
           // initialize the corresponding parameter in the instance class.
           MySoundParamInstance(inst).soundMP3 = soundMP3;
       }
    }
}
```

In the MySoundParam class, you define a variable named `soundMP3` that enables the user of the effect to specify the URL of the MP3 file to play. You also modify your override of the `initInstance()` method to pass the value of the `soundMP3` variable to the instance class.

Notice that the getAffectedProperties() method still returns an empty Array. That is because `getAffectedProperties()` returns the list of properties of the effect target that are modified by the effect, not the properties of the effect itself.

In your instance class, you define a property named soundMP3, corresponding to the property with the same name in the factory class. Then, you use it to create the URLRequest object to play the sound, as the following example shows:

```
package myEffects
{
    // createcomps_effects/myEffects/MySoundParamInstance.as
    import mx.effects.EffectInstance;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;
    public class MySoundParamInstance extends EffectInstance
    {
        // Define local variables.
        private var s:Sound;
        private var sndChannel:SoundChannel;
        private var u:URLRequest;
        // Define a variable for the MP3 URL.
        public var soundMP3:String;
        // Define constructor.
        public function MySoundParamInstance(targetObj:Object) {
            super(targetObj);
        }
        // Override play() method.
        override public function play():void     {
            // You must call super.play() from within your override.
            super.play();
            s = new Sound();
            // Use the new parameter to specify the URL.
            u = new URLRequest(soundMP3);
            s.load(u);
            sndChannel=s.play();
        }

        // Override end() method to stop the MP3.
        override public function end():void  {
            sndChannel.stop();
            super.end();
        }
    }
}
```

You can now pass the URL of an MP3 to the effect, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_effects/MainSoundParam.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComps="myEffects.*">
    <fx:Declarations>
        <MyComps:MySoundParam id="mySoundEffect"
            soundMP3="http://localhost:8100/flex/assets/sample.mp3"/>
    </fx:Declarations>
    <!-- Use the SoundEffect effect with a mouseOver trigger. -->
    <mx:Label id="myLabel"
        text="play MP3"
        rollOver="mySoundEffect.end();mySoundEffect.play();"/>
</s:Application>
```

About MX tween effects

Most MX effects are implemented by using the tweening mechanism, where a *tween* defines a transition performed on a target object over a period of time. That transition could be a change in size, such as the Zoom or Resize effects perform; a change in visibility, such as the Fade or Dissolve effects perform; or other types of transitions.

You use the following classes to implement a tween effect:

- mx.effects.Tween A class used to implement tween effects. A Tween object accepts a start value, an end value, and an optional easing function. When you define tween effect classes, you create an instance of the Tween class in your override of the `Effect.play()` method.

  The Tween object invokes the `mx.effects.TweenEffect.onTweenUpdate()` callback method on a regular interval, passing the callback method an interpolated value between the start and end values. Typically, the callback method updates some property of the target component, causing that component's property to animate over time. For example, the Move effect modifies the `x` and `y` properties of the target component for the duration of the effect to show an animated movement.

  When the effect ends, the Tween object invokes the `mx.effects.TweenEffect.onTweenEnd()` callback method. This method performs any final processing before the effect terminates. You must call `super.onTweenEnd()` from your override.

- mx.effects.TweenEffect The base factory class for all tween effects. This class encapsulates methods and properties that are common among all Tween-based effects.

- mx.effects.effectClasses.TweenEffectInstance The instance class for all tween effects.

  When you define effects based on the TweenEffect class, you must override the `TweenEffectInstance.onTweenUpdate`() method, and optionally override the `TweenEffectInstance.onTweenEnd`() method.

The following example creates a tween effect. However, Flex supplies the AnimateProperty class that you can use to create a tween effect for a single property of the target component. For more information, see "Introduction to effects" on page 1784.

Example: Creating a tween effect

In this example, you create a tween effect that rotates a component in a circle. This example implements a simplified version of the Rotate effect. The rotation is controlled by two parameters that are passed to the effect: `angleFrom` and `angleTo`:

```
package myEffects
{
   // createcomps_effects/myEffects/Rotation.as
   import mx.effects.TweenEffect;
   import mx.effects.EffectInstance;
   import mx.effects.IEffectInstance;
   public class Rotation extends TweenEffect
   {
       // Define parameters for the effect.
       public var angleFrom:Number = 0;
       public var angleTo:Number = 360;

       // Define constructor with optional argument.
       public function Rotation(targetObj:* = null) {
           super(targetObj);
           instanceClass= RotationInstance;
       }
       // Override getAffectedProperties() method to return "rotation".
       override public function getAffectedProperties():Array {
           return ["rotation"];
       }

       // Override initInstance() method.
       override protected function initInstance(inst:IEffectInstance):void {
           super.initInstance(inst);
           RotationInstance(inst).angleFrom = angleFrom;
           RotationInstance(inst).angleTo = angleTo;
       }
   }
}
```

In this example, the effect works by modifying the `rotation` property of the target component. Therefore, your override of the `getAffectedProperties()` method returns an array that contains a single element.

You derive your instance class from the TweenEffectInstance class, and override the `play()`, `onTweenUpdate()`, and `onTweenEnd()` methods, as the following example shows:

```
package myEffects
{
    // createcomps_effects/myEffects/RotationInstance.as
    import mx.effects.effectClasses.TweenEffectInstance;
    import mx.effects.Tween;
    public class RotationInstance extends TweenEffectInstance
    {
        // Define parameters for the effect.
        public var angleFrom:Number;
        public var angleTo:Number;

        public function RotationInstance(targetObj:*) {
            super(targetObj);
        }

        // Override play() method class.
        override public function play():void {
            // All classes must call super.play().
            super.play();
            // Create a Tween object. The tween begins playing immediately.
            var tween:Tween =
                createTween(this, angleFrom, angleTo, duration);
        }
        // Override onTweenUpdate() method.
        override public function onTweenUpdate(val:Object):void {
            target.rotation = val;
        }

        // Override onTweenEnd() method.
        override public function onTweenEnd(val:Object):void {
            // All classes that implement onTweenEnd()
            // must call    super.onTweenEnd().
            super.onTweenEnd(val);
        }
    }
}
```

In this example, the Tween object invokes the `onTweenUpdate()` callback method on a regular interval, passing it values between `angleFrom` and `angleTo`. At the end of the effect, the Tween object calls the `onTweenUpdate()` callback method with a value of `angleTo`. By invoking the `onTweenUpdate()` callback method at regular intervals throughout the duration of the effect, the target component displays a smooth animation as it rotates.

You use your new effect in an MXML application, as the following example shows:

```
<?xml version="1.0"?>
<!-- createcomps_effects/MainRotation.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myEffects.*"
    height="450">
    <fx:Declarations>
        <MyComp:Rotation id="Rotate90"
            angleFrom="0" angleTo="360"
            duration="1000"
            target="{myImage}"/>
    </fx:Declarations>

    <mx:Image id="myImage"
        x="250" y="200"
        source="@Embed(source='../assets/myRotationImage.jpg')"
        mouseDown="Rotate90.end();Rotate90.play();"/>
</s:Application>
```

In this example, you use the effect to rotate an image when the user clicks it.

Writing an effect for a transition

Transitions define how a change of view state appears on the screen. You define a transition by using a combination of the Flex effect classes. For more information on transitions, see "Transitions" on page 1870.

You can define your own custom effects for use in transitions. To do so, you have to account for the effect being used in a transition when you override the EffectInstance.play() method. The `EffectInstance.play()` method must be able to determine default values for effect properties when the effect is used in a transition.

Defining the default values for a transition effect

Like any effect, an effect in a transition has properties that you use to configure it. For example, most effects have properties that define starting and ending information for the target component, such as the `xFrom`, `yFrom`, `xTo`, and `yTo` properties of the Move effect.

Flex uses the following rules to determine the start and end values of effect properties when you use the effect in a transition:

1 If the effect defines the values of any properties, it uses the properties in the transition, as the following example shows:

```
<mx:Transition fromState="*" toState="*">
    <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
        <mx:Blur duration="100"
            blurXFrom="0.0" blurXTo="10.0" blurYFrom="0.0" blurYTo="10.0"/>
     <mx:Parallel>
            <mx:Move duration="400"/>
            <mx:Resize duration="400"/>
        </mx:Parallel>
     <mx:Blur duration="100"
            blurXFrom="10.0" blurXTo="0.0" blurYFrom="10.0" blurYTo="0.0"/>
    </mx:Sequence>
</mx:Transition>
```

In this example, the two Blur filters define the properties of the effect.

2 If the effect does not define the start values of the effect, the effect determines the values from the EffectInstance.propertyChanges property passed to the effect instance. Flex sets the `propertyChanges` property by using information from the current settings of the component, as defined by the current view state. For more information on the `propertyChanges` property, see "How Flex initializes the propertyChanges property" on page 2536.

In the example in step 1, notice that the Move and Resize effects do not define start values. Therefore, Flex determines the start values from the current size and position of the effect targets in the current view state, and passes that information to each effect instance by using the `propertyChanges` property.

3 If the effect does not define the end values of the effect, the effect determines the values from the `Effectinstance.propertyChanges` property passed to the effect instance. Flex sets the `propertyChanges` property by using information about the component, as defined by the destination view state. For more information on the `propertyChanges` property, see "How Flex initializes the propertyChanges property" on page 2536.

In the example in rule 1, Flex determines the end values of the Move and Resize effects from the size and position of the effect targets in the destination view state. In some cases, the destination view state defines those values. If the destination view state does not define the values, Flex determines them from the setting of the base view state, and passes that information to each effect instance by using the `propertyChanges` property.

4 If there are no explicit values, and Flex cannot determine values from the current or destination view states, the effect uses its default property values.

Using the propertyChanges property

The EffectInstance.propertyChanges property contains a PropertyChanges object. A PropertyChanges object contains the properties described in the following table:

Property	Description
`target`	A target component of the effect. The `end` and `start` properties of the PropertyChanges class define how the target component is modified by the change to the view state.
`start`	An object that contains the starting properties of the `target` component, as defined by the current view state. For example, for a `target` component that is moved and resized by a change to the view state, the `start` property contains the starting position and size of the component, as the following example shows: `{x:00, y:00, width:100, height:100}`
`end`	An object that contains the ending properties of the `target` component, as defined by the destination view state. For example, for a `target` component that is moved and resized by a change to the view state, the `end` property contains the ending position and size of the component, as the following example shows: `{x:100, y:100, width:200, height:200}`

In the body of the `Effectinstance.play()` method, you can examine the information in the `propertyChanges` property to configure the effect.

How Flex initializes the propertyChanges property

Before you can use the EffectInstance.propertyChanges property in your effect instance, it has to be properly initialized. When you change the view state, Flex initializes `propertyChanges.start` and `propertyChanges.end` separately.

The following steps describe the actions that occur when you change view states. Notice that Flex initializes `propertyChanges.start` as part of step 4, and initializes `propertyChanges.end` as part of step 7.

1 You set the `currentState` property to the destination view state.

2. Flex dispatches the `currentStateChanging` event.
3. Flex examines the list of transitions to determine the one that matches the change of view state.
4. Flex calls the `Effect.captureStartValues()` method to initialize `propertyChanges.start` for all effect instances.

   You can also call `Effect.captureStartValues()` to initialize an effect instance used outside of a transition.
5. Flex applies the destination view state to the application.
6. Flex dispatches the `currentStateChange` event.
7. Flex plays the effects defined in the transition.

   As part of playing the effect, Flex invokes the factory class play() method and the `Effect.play()` method to initialize `propertyChanges.end` for effect instances.

Example: Creating a transition effect

The following example modifies the Rotation effect created in “Example: Creating a tween effect” on page 2532 to make the effect usable in a transition.

This example shows the RotationTrans.as class that defines the factory class for the effect. To create RotationTrans.as, the only modification you make to Rotation.as is to remove the default value definitions for the `angleFrom` and `angleTo` properties. The `RotationTransInstance.play()` method determines the default values.

```
package myEffects
{
   // createcomps_effects/myEffects/RotationTrans.as
   import mx.effects.TweenEffect;
   import mx.effects.EffectInstance;
   import mx.effects.IEffectInstance;
   public class RotationTrans extends TweenEffect
   {
       // Define parameters for the effect.
       // Do not specify any default values.
       // The default value of these properties is NaN.
       public var angleFrom:Number;
       public var angleTo:Number;

       // Define constructor with optional argument.
       public function RotationTrans(targetObj:Object = null) {
           super(targetObj);
           instanceClass= RotationTransInstance;
       }
       // Override getAffectedProperties() method to return "rotation".
       override public function getAffectedProperties():Array {
           return ["rotation"];
       }

       // Override initInstance() method.
       override protected function initInstance(inst:IEffectInstance):void {
           super.initInstance(inst);
           RotationTransInstance(inst).angleFrom = angleFrom;
           RotationTransInstance(inst).angleTo = angleTo;
       }
   }
}
```

In the RotationTransInstance.as class, you modify the `play()` method to calculate the default values for the `angleFrom` and `angleTo` properties. This method performs the following actions:

1 Determines whether the user set values for the `angleFrom` and `angleTo` properties.

2 If not, determines whether the `Effectinstance.propertyChanges` property was initialized with start and end values. If so, the method uses those values to configure the effect.

3 If not, sets the `angleFrom` and `angleTo` properties to the default values of 0 for the `angleFrom` property, and 360 for the `angleTo` property.

The following example shows the RotationTransInstance.as class:

```
package myEffects
{
    // createcomps_effects/myEffects/RotationTransInstance.as
    import mx.effects.effectClasses.TweenEffectInstance;
    import mx.effects.Tween;
    public class RotationTransInstance extends TweenEffectInstance
    {
        // Define parameters for the effect.
        public var angleFrom:Number;
        public var angleTo:Number;

        public function RotationTransInstance(targetObj:Object) {
            super(targetObj);
        }

        // Override play() method class.
        override public function play():void {
            // All classes must call super.play().
            super.play();

            // Check whether angleFrom is set.
            if (isNaN(angleFrom))
            {
              // If not, look in propertyChanges.start for a value.
              // Otherwise, set it to 0.
              angleFrom = (propertyChanges.start["rotation"] != undefined) ?
                    propertyChanges.start["rotation"] : 0;
            }

            // Check whether angleTo is set.
            if (isNaN(angleTo))
            {
                // If not, look in propertyChanges.end for a value.
                // Otherwise, set it to 360.
                angleTo = (propertyChanges.end["rotation"] != undefined) ?
```

```
                    propertyChanges.end["rotation"] : 360;
            }

            // Create a Tween object. The tween begins playing immediately.
            var tween:Tween =
                createTween(this, angleFrom, angleTo, duration);
        }
        // Override onTweenUpdate() method.
        override public function onTweenUpdate(val:Object):void {
            target.rotation = val;
        }

        // Override onTweenEnd() method.
        override public function onTweenEnd(val:Object):void {
            // All classes that implement onTweenEnd()
            // must call super.onTweenEnd().
            super.onTweenEnd(val);
        }
    }
}
```

The following application uses the RotationTrans effect:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- createcomps_effects/MainRotationTrans.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myEffects.*">
    <!-- Define the two view states, in addition to the base state.-->
    <s:states>
        <s:State name="Default"/>
        <s:State name="One"/>
        <s:State name="Two"/>
    </s:states>

    <!-- Define the single transition for all view state changes.-->
    <s:transitions>
        <mx:Transition fromState="*" toState="*">
            <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
                <mx:Parallel>
                    <mx:Move  duration="400"/>
                    <mx:Resize duration="400"/>
                </mx:Parallel>
                <MyComp:RotationTrans filter="move"/>
            </mx:Sequence>
        </mx:Transition>
    </s:transitions>

    <!-- Define the Canvas container holdig the three Panel containers.-->
    <mx:Canvas id="pm" width="100%" height="100%" >
        <mx:Panel id="p1" title="One"
                x="0" y="0"
                x.One="110"
                width="100" height="100"
                width.One="200" height.One="200"
                click="currentState='One'" >
            <mx:Label fontSize="24" text="One"/>
        </mx:Panel>
        <mx:Panel id="p2" title="Two"
                x="0" y="110"
                y.One="0"
                x.Two="110" y.Two="0"
                width="100" height="100"
                width.Two="200" height.Two="210"
                click="currentState='Two'" >
            <mx:Label fontSize="24" text="Two"/>
        </mx:Panel>
        <mx:Panel id="p3" title="Three"
                x="110" y="0"
                x.One="0" y.One="110"
                x.Two="0" y.Two="110"
                width="200" height="210"
                width.One="100" height.One="100"
                width.Two="100" height.Two="100"
                click="currentState=''" >
            <mx:Label fontSize="24" text="Three"/>
        </mx:Panel>
    </mx:Canvas>
</s:Application>
```

Defining a custom effect trigger for MX effects

You can create a custom effect trigger to handle situations for which the standard Flex triggers do not meet your needs. An effect trigger is paired with a corresponding event that invokes the trigger. For example, a Button control has a `mouseDown` event and a `mouseDownEffect` trigger. The event initiates the corresponding effect trigger when a user clicks a component. You use the `mouseDown` event to specify the event listener that executes when the user selects the component. You use the `mouseDownEffect` trigger to associate an effect with the trigger.

Suppose that you want to apply an effect that sets the brightness level of a component when a user action occurs. The following example shows a custom Button control that uses a new property, `bright`, and dispatches two new events, `darken` and `brighten`, based on changes to the `bright` property. The control also defines two new effect triggers, `darkenEffect` and `brightenEffect`, which are paired with the `darken` event and the `brighten` event.

```
<?xml version="1.0"?>
<!-- createcomps_effects\myComponents\MyButton.mxml -->
<mx:Button xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
    <fx:Metadata>
        <!-- Define the metadata for the events and effect triggers. -->
        [Event(name="darken", type="flash.events.Event")]
        [Event(name="brighten", type="flash.events.Event")]
        [Effect(name="darkenEffect", event="darken")]
        [Effect(name="brightenEffect", event="brighten")]
    </fx:Metadata>

    <fx:Script>
        <![CDATA[
            import flash.events.Event;
            // Define the private variable for the bright setting.
            private var _bright:Boolean = true;

            // Define the setter to dispatch the events
            // corresponding to the effect triggers.
            public function set bright(value:Boolean):void {
                _bright = value;

                if (_bright)
                    dispatchEvent(new Event("brighten"));
                else
                    dispatchEvent(new Event("darken"));
            }

            // Define the getter to return the current bright setting.
            public function get bright():Boolean {
                return _bright;
            }
        ]]>
    </fx:Script>
</mx:Button>
```

When you declare an event in the form `[Event(name="`*eventName*`", type="`*package.eventType*`")]`, you can also create a corresponding effect, in the form `[Effect(name="`*eventname*`Effect", event="`*eventname*`")]`. As in the previous example, in the `<fx:Metadata>` tag, you insert the metadata statements that define the two new events, `darken` and `brighten`, and the new effect triggers, `darkenEffect` and `brightenEffect`, to the Flex compiler.

For more information on using metadata, see "Metadata tags in custom components" on page 2376.

The application in the following example uses the MyButton control. The `darkenEffect` and `brightenEffect` properties are set to the FadeOut and FadeIn effects, respectively. The `click` event of a second Button control toggles the MyButton control's `bright` property and executes the corresponding effect (FadeOut or FadeIn).

```
<?xml version="1.0"?>
<!-- createcomps_effects/MainMyButton.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:MyComp="myComponents.*" >
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Define two fade effects for darkening and brightening target. -->
        <mx:Fade id="FadeOut"
            duration="1000"
            alphaFrom="1.00" alphaTo=".20"/>
        <mx:Fade id="FadeIn"
            duration="1000"
            alphaFrom=".20" alphaTo="1.00"/>
    </fx:Declarations>
    <!-- Define custom button that defines the
        darkenEffect and brightenEffect. -->
    <MyComp:MyButton
        label="MyButton" id="btn"
        darkenEffect="{FadeOut}"
        brightenEffect="{FadeIn}"
        darken="debugW.text='got darken event';"
        brighten="debugW.text='got brighten event';"/>

    <!-- Define button that triggers darken event. -->
    <mx:Button
        label="set bright to false"
        click="btn.bright = false; myTA.text=String(btn.bright);"/>

    <!-- Define button that triggers brighten event. -->
    <mx:Button
        label="set bright to true"
        click="btn.bright = true; myTA.text=String(btn.bright);"/>

    <!-- TextArea displays the current value of bright. -->
    <mx:TextArea id="myTA" />

    <!-- TextArea displays event messages. -->
    <mx:TextArea id="debugW" />
    <!-- Define button to make sure effects working. -->
    <MyComp:MyButton id="btn2" label="test effects"
        mouseDownEffect="{FadeOut}"
        mouseUpEffect="{FadeIn}"/>
</s:Application>
```

Overriding the initEffect() method

The EffectInstance class defines the initEffect() method that you can override in your custom effect. This method has the following signature:

```
public initEffect(event:Event):void
```

where *event* is the Event object dispatched by the event that triggered the effect.

For example, a user might create an instance of an effect, but not provide all of the configuration information that is required to play the effect. Although you might be able to assign default values to all properties within the definition of the effect class, in some cases you might have to determine the default values at run time.

In this method, you can examine the event object and the effect target to calculate values at run time. For more information on how to create a custom event and an effect trigger, see “Defining a custom effect trigger for MX effects” on page 2540. As part of that example, you can add properties to the event object passed to the `dispatchEvent()` method. You can then access that event object, and its additional properties, from the `initEffect()` method.

By overriding the `initEffect()` method, you can also access the `target` property of the Event object to reference the target component of the effect. For example, if you must determine the current x and y coordinates of the component, or its current height and width, you can access them from your override of the `initEffect()` method.

Chapter 11: Deploying applications

Deploying applications

When you deploy an application written in Adobe® Flex®, you make the application accessible to your users. The process of deploying an application is dependent on your application, your application requirements, and your deployment environment. For example, the process of deploying an application on an internal website that is only accessible by company employees might be different from the process for deploying the same application on a public website accessible by anyone.

Use the overview of the deployment process and the general checklist as a guide when you deploy your application. This topic does not attempt to define the exact set of steps that you use for deploying all applications. Instead, it contains an overview of the deployment process, and a general checklist that you might use when you deploy your application.

About deploying an application

When you deploy an application, you move the application from your development environment to your deployment environment. After you deploy it, customers have full access to the application.

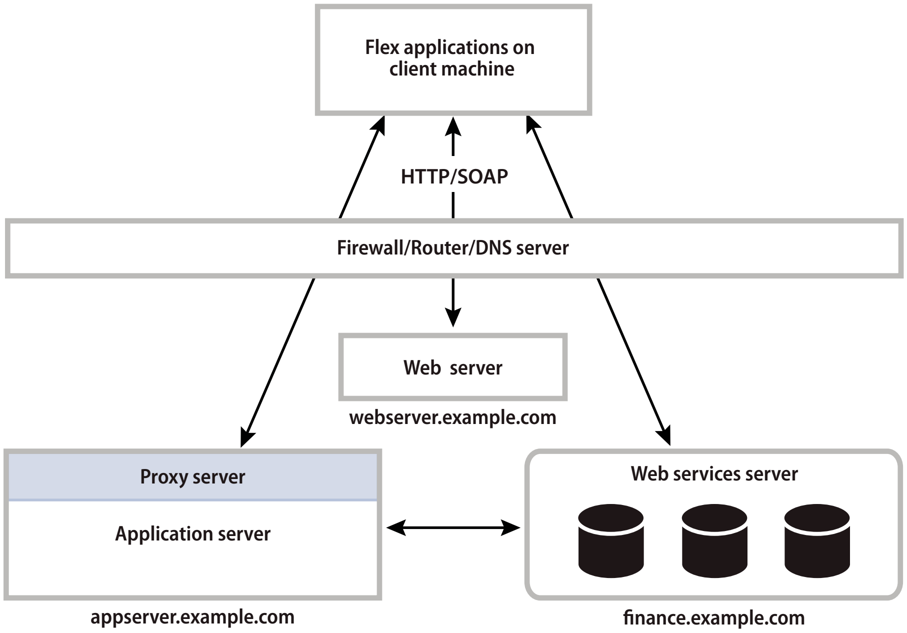
The deployment process that your organization uses might only require you to copy a application's SWF file from your development server to your deployment server. In many organizations however, the deployment process is more complicated, and involves people from groups outside the development organization. For example, you might have an IT department that maintains your corporate website. The IT department might be responsible for staging, testing, and then deploying your application.

Your application architecture might also require you to deploy more than just a single SWF file. For example, your application might access Runtime Shared Libraries (RSLs) or other assets at run time. You must make sure to copy all required files to your deployment environment.

Deployment might also require you to perform operations other than just copying application files to a deployment server. Your application might access data services on your server, or on another server. You must ensure that your data services are accessible by a deployed Flex application that executes on a client's computer.

Deployment options

The following example shows a typical deployment environment for an application:



Deploying an application for Flex SDK and Adobe® Flex® Builder® - might require you to perform some or all of the following actions:

- Copy the application SWF file to your deployment server. As the previous example shows, you copy the application to webserver.example.com.
- Copy any asset files, such as icons, media files, or other assets, to your deployment server.
- Copy any custom RSLs to your web server or application server. For more information, see "Deploying RSLs with Flex SDK" on page 2546.
- Copy the framework RSLs to your deployment server. For more information, see "Using the framework RSLs" on page 260.
- Copy any SWF files for your module to your deployment server in the same directory structure as you used for your development environment. "Deploying modules with Flex SDK" on page 2546.
- Copy any SWF files required to support Flex features, such as deep linking or runtime CSS. For more information, see "Deploying additional Flex files" on page 2546.
- Write a wrapper for the SWF file if you access it from an HTML, JSP, ASP, or another type of page.

  A deployed SWF file can encompass your entire web application, however it is often used as a part of the application. Therefore, users do not typically request the SWF file directly, but request a web page that references the SWF file. Flash Builder can generate the wrapper for you, or, you can write the wrapper. For more information, see "Creating a wrapper" on page 2552.
- Create a crossdomain.xml file on the server for data service, if you directly access any data services outside of the domain that serves the SWF file. For more information, see "Accessing data services from a deployed application" on page 2546

Deploying RSLs with Flex SDK

When your application uses custom RSLs, you must make sure to deploy the RSL on your deployment server, in the same domain, unless you are using cross-domain RSLs. You use the `runtime-shared-libraries` option of the Flex compiler to specify the directory location of the RSL at compile time. Ensure that you copy the RSL to the same directory that you specified with `runtime-shared-libraries`.

By default, the Flex compilers dynamically link your application against the framework RSLs. This means that classes in the SDK are externally loaded at run time from the framework RSLs. The default location of these RSLs is on the Adobe web site. If you clients do not have network connectivity, you must deploy the framework RSLs to a local location, or disable framework RSLs. To customize the location of framework RSLs, edit the flex-config.xml file or edit their location in the Flash Builder Flex Build Path dialog box. To disable framework RSLs, set the `static-link-runtime-shared-libraries` compiler option to `true`.

Flex also provides framework RSLs. These libraries are comprised of the Flex class libraries and can be used with any application built with Flex. The framework RSLs are precompiled libraries of framework classes and components. If your client does not have internet connectivity, be sure to deploy both the signed (*.SWZ) and unsigned (*.SWF) RSLs. Flash Player will first try to load the signed RSLs from the Adobe web site, so you should not have to deploy them in most cases.

For more information, see “Runtime Shared Libraries” on page 253.

Deploying modules with Flex SDK

Modules are SWF files that can be loaded and unloaded by an application. They cannot be run independently of an application, but any number of applications can share the modules. When your application uses a module, you must make sure to deploy the module’s SWF file on your deployment server in the same directory structure as you used for your development environment. For more information, see “Modular applications” on page 138.

Deploying additional Flex files

The implementation of some Flex features requires that you deploy additional files along with your application’s SWF file. For example, if you use deep linking functionality in your application, you must deploy the historyFrame.html, history.css, and history.js files. If you use the Express Install version detection feature, you also must deploy the playerProductInstall.swf file with your SWF file. You typically deploy these files in the same location that the default HTML wrapper looks for them. For example, the deep linking files are typically in a sub directory called /history.

For a complete list of additional Flex files that you might deploy with your application, see “Deployment checklist” on page 2548.

Accessing data services from a deployed application

In a typical Flex development environment, you build and test your application behind a corporate firewall, where security restrictions are much less strict than when a customer runs the application on their own computer. However, when you deploy the application, it runs on a customers computer outside your firewall. That simple change of location might cause the application to fail if you do not correctly configure your data services to allow external access.

Most run-time accesses to application resources fall into one of the following categories:

- Direct access to asset files on a web server, such as image files.
- Direct access to resources on your J2EE application server.
- Data services requests through a proxy. A proxy redirects that request to the server that handles the data service request.
- Direct access to a data service.

As part of deploying your application, ensure that all run-time data access requests work correctly from the application that is executing outside of your firewall.

Compiling for deployment

When you create a deployable SWF file, ensure that you compile the application correctly. Typically, you disable certain compiler features, such as the generation of debug output, and enable other options, such as the generation of accessible content.

This section contains an overview of some common compiler options that you might use when you create a deployable SWF file. For a complete list of compiler options, see "Flex compilers" on page 2164.

Creating a release build of your application in Flash Builder

When you create and run an application in Flash Builder, the Flex compiler includes debug information in that application so that you can set breakpoints and view variables and perform other debugging tasks. However, the SWF file that Flash Builder generates by default includes debugging information that makes it larger than the release build of the SWF file.

To create a release build of your application in Flash Builder, select Project > Export Release Build. This compiles a version of your application's SWF file that does not contain any debug information in it. This SWF file is smaller than the SWF files you compile normally. This also compiles any modules that are in the application's project without debug information.

In general, all compiled SWF files that are stored in the project's /bin-debug directory contain debug information. When you export the application, you choose a new output directory. The default is the /bin-release directory.

To compile a release build on the command line, set the `debug` compiler option to `false`. This prevents debug information from being included in the final SWF file.

Enabling accessibility

The Flex accessibility option lets you create applications that are accessible to users with disabilities. By default, accessibility is disabled. You enable the accessibility features of Flex components at compile time by using options to a Flash Builder project, setting the `accessible` option to `true` for the command-line compiler, or setting the `<accessible>` tag in the flex-config.xml file to `true`.

For more information on creating accessible applications, see "Accessible applications" on page 2122.

Preventing users from viewing your source code

Flex lets you publish your source code with your deployed application. You might want to enable this option during the development process, but disable it for deployment. Or, you might want to include your source code along with your deployed application.

In Flash Builder, the Export Release Build wizard lets you specify whether to publish your source code. You can also use the `viewSourceURL` property of the Application class to set the URL of your source code.

Disabling incremental compilation

You can use incremental compilation to decrease the time it takes to compile an application or component library with the Flex application compilers. When incremental compilation is enabled, the compiler inspects changes to the bytecode between revisions and only recompiles the section of bytecode that has changed.

For more information, see "Flex compilers" on page 2164.

Using a headless server

A *headless server* is one that is running UNIX or Linux and often does not have a monitor, keyboard, mouse, or even a graphics card. Headless servers are most commonly encountered in ISPs and ISVs, where available space is at a premium and servers are often mounted in racks. Enabling the headless mode reduces the graphics requirements of the underlying system and can allow for a more efficient use of memory.

If you deploy a Flex application on a headless server, you must set the `headless-server` option of the compiler to `true`. Setting this option to `true` is required to support fonts and SVG images in a nongraphical environment.

Deployment checklist

The deployment checklist contains some common system configuration issues that customers have found when deploying Flex applications for production. It also contains troubleshooting tips to diagnose common deployment problems.

Application assets

When deploying a Flex application, you must make sure you also deploy all the assets that the application uses at run time. These include files that are used by the wrapper to support features such as deep linking and Express Install, as well as files that are loaded by the application such as resource bundles or RSLs.

In the case of wrapper code, you will probably be cutting and pasting it from the HTML template included with the SDK into your JSP or ASP or PHP pages.

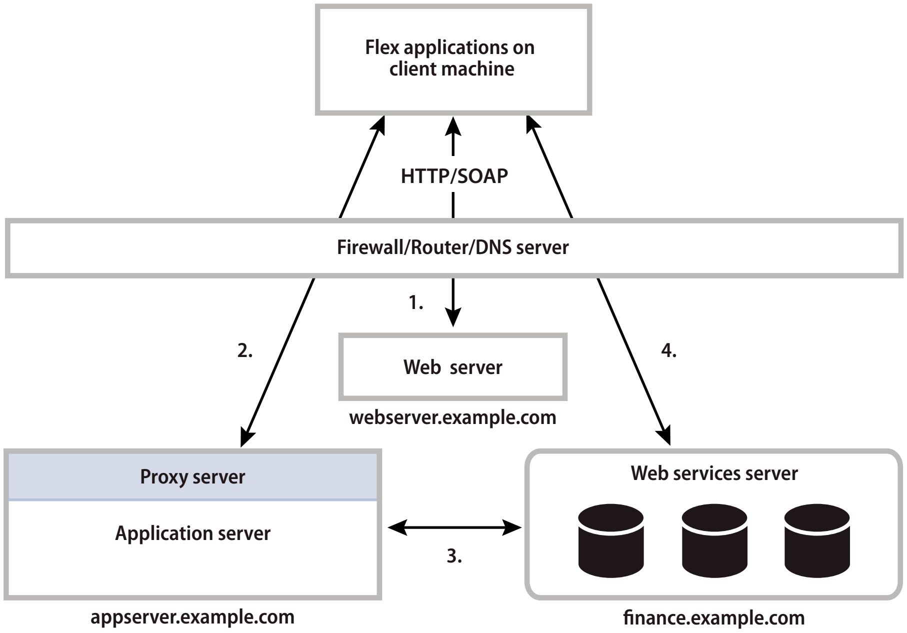
Check that the following assets are deployed with your application if you use those assets in your Flex applications:

Feature	Assets to Deploy
Wrapper files	If you use a wrapper, be sure to include it in the deployment process. The wrapper can be any file that returns HTML, such as PHP, ASP, JSP, or ColdFusion. Typically, this file uses the SWFObject 2 logic, or includes an `<object>` or `<embed>` tag to embed the Flex application. In addition, if you use dynamic pages to query databases or perform other server-side actions for your Flex application, be sure to deploy those as well. This is especially important if you use the data wizard in Flash Builder to generate these pages. For more information, see "Creating a wrapper" on page 2552.
Version detection	To support version detection or Express Install in your HTML wrapper, you must add the code based on the Flex wrapper template to your wrapper, as well as deploy the swfobject.js file. You must also deploy the playerProductInstall.swf file. For more information, see "Creating a wrapper" on page 2552.
Deep linking	To support deep linking, you must include the following files in your deployment: • history.js • history.css • historyFrame.html You must import the first two files into your HTML wrapper, and store all of these files in a /history subdirectory. For more information, see "Deep linking" on page 2022.

Feature	**Assets to Deploy**
Runtime shared libraries (RSLs)	For standard RSLs, deploy the RSL SWF files with your Flex application. You must be sure to deploy the SWF files to the same relative location that the compiler used. If you are deploying an a custom RSL, be sure to optimize the RSL's SWF file prior to deployment. For framework RSLs, if your client does not have internet connectivity, be sure to deploy both the signed (*.SWZ) and unsigned (*.SWF) RSLs. Flash Player will first try to load the signed RSLs from the Adobe web site, so you should not have to deploy them in most cases. For framework and cross-domain RSLs, be sure to deploy failover RSLs to the locations you specified when you compiled the application. For more information, see "Runtime Shared Libraries" on page 253.
Runtime stylesheets	If you use runtime stylesheets in your application, you must deploy the SWF files so that they can be loaded. You can load run-time stylesheet SWF files either locally or remotely. However, if you load them locally, the stylesheets must be in the same relative location that you specified in the application. For more information, see "Loading style sheets at run time" on page 1547.
Modules	If your application uses modules, you must deploy the module SWF files so that they can be loaded. Modules are SWF files that can be loaded and used by any number of applications. If multiple applications use your modules, then you should deploy them to a location that all applications can load them from rather than deploy them multiple times across different domains. For more information, see "Modular applications" on page 138.
Runtime localization	If your application uses run-time localization (if it, for example, lets the user switch from English to Japanese language at run time), then you might need to deploy resource module SWF files. These modules can contain one or more resources bundles for one or more locales. For more information, see "Localization" on page 2055.
Security files	If you use container-based security, then be sure to update your security constraints to include your Flex application. In addition, if you load assets from multiple domains, be sure to deploy any crossdomain.xml files that are required by your applications.
Miscellaneous runtime assets	Not all assets are embedded at compile time. For example, FLV and image files are usually loaded at run time to keep the SWF file as small as possible. Be sure to check that you deploy the following types of assets that are typically loaded at run time with your Flex application: • FLV files • SWF files • Sound files (such as MP3 files) • Images (such as GIF, JPG, and PNG files)
Data files	It is not uncommon for flat data files to be used as a data provider in Flex applications. Be sure to deploy any text files, which might appear in various formats such as XML, that are loaded at run time.
View source	If you use the view source functionality in Flash Builder, be sure to include the supporting files when you deploy your application. These files include the selected source code files, the source viewer's SWF file, HTML and CSS files for the source viewe, an XML file, and a ZIP file of the source code that users can download. You must maintain the directory structure that Flash Builder generates in the output directory. To enable view source and generate the ZIP file in Flash Builder, select Project > Export Release Build. Then select the Enable View Source option. For more information, see Publish source code.

Types of network access

Deployed applications typically make several types of requests to services within your firewall, as the following example shows:



Most of the deployment issues that customers report are related to network security and routing, and fall into one of the following scenarios:

1. Direct access to resources on a web server, such as image files. In the preceding example, the client directly accesses resources on webserver.example.com.
2. Direct access to resources on your application server. In the preceding example, the client directly accesses resources on appserver.example.com. Ensure that deployed applications can access the appropriate servers.
3. Web services requests through a proxy. A proxy redirects a request to the server that handles the web service. In the preceding example, the client accesses a resource on appserver.example.com, but that request is redirected to finance.example.com. Ensure that you configure the proxy server correctly so that deployed Flex applications can access your web services, or other data services, through the proxy.
4. Direct access of a web service. In the preceding example, the client directly accesses a service on finance.example.com. If a deployed Flex application directly accesses web services, or other data services, ensure that access is allowed.

Step 1. Create a list of server-side resources

Before you start testing your network configuration, make a list of the IP addresses and DNS names of all the servers that a Flex application might access. A Flex application might directly access these servers, for example by using a web service, or another server might access them as part of handling a redirected request.

Enter the information about your servers in the following table:

Name	DNS name	IP address

Enter information about the server hosting the web service proxy:

Name	DNS Name	IP Address

Enter information about your web services or any other services accessible from a deployed Flex application:

Name	Location (URL)

Step 2. Verify access from server to server within your firewall

In some cases, an external request to one server can be redirected to another server behind your firewall. A redirected request can occur for a request to a web service or to any file, depending on the system configuration. Where it is necessary, ensure that your servers can communicate with each other so that a redirected request can be properly handled.

To determine if one server, called Server A in this example, can communicate with another server, called Server B, create a temporary file called temp.htm on Server A in its web root directory. Then, log in to Server B and ensure that it can access temp.htm on Server A. Try to access the file by using Server A's DNS name and also its IP address.

Servers can have multiple NIC cards or multiple IP addresses. Ensure that each server can communicate with all of the IP addresses on your other servers.

Also, log in to the server that hosts your web service proxy to make sure that it can access all web services on all other servers. You can test the web service proxy by making an HTTP request to the WSDL file for each web service. In the previous example, log in to appserver.example.com and ensure that it can access the WSDL files on finance.example.com.

If any server cannot access the other servers in your system, an external request from a Flex application might also fail. For more information, contact your system administrator.

Step 3. Verify access to your servers from outside the firewall

Some servers might have to be accessed from outside the firewall to handle HTTP, SOAP, or AMF requests from clients. You can use the following methods to determine if a deployed Flex application can access your servers from outside the firewall:

1. On each server that can be accessed from outside the firewall, create a temporary file, such as temp.htm, on the server in its web root directory. From a computer outside the firewall, use a browser to make an HTTP request to the temporary file to ensure that an external computer can access it.

   For example, for a file named temp.htm, try accessing it by using the following URL:

   ```
   http://webserver.example.com/server1/temp.htm
   ```

2. From a computer outside the firewall, use a browser to make an HTTP request to the WSDL file for each web service that can be accessed from outside the firewall to ensure that the WSDL file can be accessed.

   For example, try accessing the WSDL file for a web service by using the following URL:

   ```
   http://finance.example.com/server1/myWS.wsdl
   ```

You should be able to access the temp.htm file or the WSDL file on all of your servers from outside the firewall. If these requests fail, contact your IT department to determine why the files cannot be accessed.

Step 4. Configure the proxy server

In “Step 3. Verify access to your servers from outside the firewall” on page 2552, you ensure that you can directly access your servers and server resources from outside the firewall.

After you configure your proxy server, ensure that the deployed Flex application can access web services and other server-side resources as necessary.

Step 5. Create a crossdomain policy file

Your system might be configured to allow a Flex application to directly access server-side resources on different domains or different computers without going through a proxy. These operations fail under the following conditions:

- When the Flex application’s SWF file references a URL, and that URL is outside the exact domain of the SWF file that makes the request
- When the Flex application’s SWF file references an HTTPS URL, and the SWF file that makes the request is not served over HTTPS

To make a data service or asset available to SWF files in different domains or on different computers, use a crossdomain policy file on the server that hosts the data service or asset. A *crossdomain policy file* is an XML file that provides a way for the server to indicate that its data services and assets are available to SWF files served from certain domains, or from all domains. Any SWF file that is served from a domain specified by the server’s policy file is permitted to access a data service or asset from that server. By default, place the crossdomain.xml at the root directory of the server that is serving the data.

For more information on using a cross-domain policy file, see “Using cross-domain policy files” on page 125.

Creating a wrapper

You typically embed an application built with Flex™ as a SWF file in an HTML page. The HTML page often includes script or external files that provide other functionality. Collectively, the HTML page and other external files are known as the *wrapper*.

About the wrapper

A simple wrapper is responsible for embedding the application's SWF file in a web page, such as an HTML, ASP, JSP, or Adobe ColdFusion page. In more complex wrappers, you can enable features such as deep linking and Express Install. The wrapper can also ensure that users both with and without JavaScript enabled in their browsers can access your applications built with Flex. You can also use the wrapper to pass `flashVars` variables into your applications and to use the ExternalInterface API. These topics are described in "Communicating with the wrapper" on page 226.

There are several ways to create a wrapper:

- Write a simple wrapper using the instructions in "Create a custom wrapper" on page 2559.
- Generate an HTML wrapper in Flash Builder. This wrapper includes features such as deep linking and Express Install support by default.
- Use the `html-wrapper` Flex Ant task. For more information, see "Using the html-wrapper task" on page 2350.

The mxmlc command-line compiler does not generate a wrapper. When using mxmlc, you generally write the wrapper manually or use the existing template as a guide. You can start out with a simple wrapper that just embeds your application's SWF file. You can then add features such as deep linking and Express Install support to your wrapper.

About the Flash Builder wrapper

Adobe® Flash® Builder™ generates a wrapper that embeds your application built with Flex. The Flash Builder wrapper includes support for Express Install and deep linking by default, although you can disable these features or configure them to your specifications.

To view the wrapper generated by Flash Builder, run the current project. Flash Builder generates an HTML page in the same location as the project's SWF file output. This directory also includes the wrapper's supporting files such as the swfobject.js and playerProductInstall.swf files.

You can configure the wrapper by using the Flex Compiler properties dialog box in Flash Builder. Configuration settings include:

- Enable or disable wrapper generation
- Set the minimum required version of Flash Player
- Use Express Install
- Enable deep linking support

For more information, see Flex compiler options.

About the HTML template

Flex SDK includes an HTML template and supporting files in the *flex_install_dir*/templates/swfobject directory. For Flash Builder, these files are located in the *install_dir*/sdks/4.6.0/templates/swfobject directory. The file name is index.template.html.

For clients with scripting enabled, the template uses SWFObject 2 to embed the SWF file built with Flex. For clients with scripting disabled, the template provides a `<noscript>` section that uses the `<object>` and `<embed>` tags to embed the SWF file.

For deployment, rename the template to index.html or whatever filename fits your web site's configuration. If you already have the HTML set up for your web site and are incorporating applications built with Flex into that site, then you can copy and paste the code from the template into your existing web site's files. The template HTML also works with dynamic scripting code such as PHP, ASP, or JSP.

The template uses tokens, such as `${height}` and `${title}`. Flash Builder replaces these tokens with the appropriate values when it compiles a project. If you are editing the wrapper manually and deploying an application built with the SDK, then you manually replace these tokens with the appropriate values.

In many cases, the tokens set the values of parameters that are passed to the `swfobject.embedSWF()` JavaScript method, or are used in the `<object>` and `<embed>` tags in the `<noscript>` block of the template.

The following table describes the template tokens:

Token	Description
`${application}`	Identifies the SWF file to the host environment (a web browser, typically) so that it can be referenced by using a scripting language. This token sets the value of the `attributes.id` and `attributes.name` properties in the SWFObject 2 logic.
`${bgcolor}`	Specifies the background color of the application. Use this property to override the background color setting specified in the application. This property does not affect the background color of the HTML page. Valid formats for `bgcolor` are any #RRGGBB, hexadecimal, or RGB value. This token sets the value of the `params.bgcolor` property in the SWFObject 2 logic.
`${expressInstallSwf}`	Enables Express Install for the embedded SWF file. To enable Express Install, set this property to "playerProductIntsall.swf". To disable Express Install, set this property to an empty string. If you enable Express Install, you must also deploy this SWF file with your application built with Flex. This token sets the value of the `xiSwfUrlStr` property in the SWFObject 2 logic. To toggle Express Install in Flex Builder, use the Use Express Install option on the Flex Compiler Properties dialog box.
`${height}`	Defines the height, in pixels, of the SWF file. Flash Player makes a best guess to determine the height of the application if none is provided. The browser scales an object or image to match the height and width specified by the author. You can set this value to a fixed number or a percentage value; for example, `'100'` or `'50%'`. Lengths expressed as percentages are based on the horizontal or vertical space currently available, not on the default size of the SWF file. You can also set the height of an application by setting the `height` property of the `<s:Application>` tag in an MXML file. This token sets the value of the `heightStr` property in the SWFObject 2 logic.
`${swf}`	Specifies the location of the SWF file. This token sets the value of the `swfUrlStr` property in the SWFObject 2 logic.
`${title}`	The title of the HTML page. This value appears in the browser's title bar when the user requests the HTML page. The default value supplied by Flash Builder is the name of the application.
`${useBrowserHistory}`	Enables deep linking for the embedded SWF file. To enable deep linking, set this property to "--". To disable deep linking, remove the token and the `<script>` tags that follow it. If you enable deep linking, youmust also deploy the files in the /templates/swfobject/history file with your application built with Flex. To toggle deep linking in Flex Builder, use the Enable Integration with Browser Navigation option on the Flex Compiler Properties dialog box.

Token	Description
`${version_major}`	The required major version number of Flash Player. For example, 10. This token is one part of the `swfVersionStr` property in the SWFObject 2 logic. It is used for Express Install. The value of this token is set in Flash Builder's Flex Compiler Properties dialog box.
`${version_minor}`	The required minor version number of Flash Player. For example, 0. This token is one part of the `swfVersionStr` property in the SWFObject 2 logic. It is used for Express Install. The value of this token is set in Flash Builder's Flex Compiler Properties dialog box.
`${version_revision}`	The required revision version number of Flash Player. For example, 162. This token is one part of the `swfVersionStr` property in the SWFObject 2 logic. It is used for Express Install. The value of this token is set in Flash Builder's Flex Compiler Properties dialog box.
`${width}`	Defines the width, in pixels, of the SWF file. Flash Player makes a best guess to determine the width of the application if none is provided. Browsers scale an object or image to match the height and width specified by the author. You can set this value to a fixed number or a percentage value. For example, `'100'` or `'50%'`. Lengths expressed as percentages are based on the horizontal or vertical space currently available, not on the natural size of the SWF file. You can also set the width of an application by setting the `width` property of the `<s:Application>` tag in an MXML file. This token sets the value of the `widthStr` property in the SWFObject 2 logic.

About SWFObject 2

SWFObject 2 is a standards-based library that embeds SWF files in HTML pages. It abstracts implementation details about Plugin detection, embedding, and other features so that you only need to call a single method to embed your SWF file. The default template included with Flex SDK and Flash Builder embeds the SWFObject 2 functionality with the following `<script>` tag:

```
<script type="text/javascript" src="swfobject.js"></script>
```

To embed a SWF file built with Flex, the HTML wrapper creates a number of properties and objects, and then passes them to the `swfobject.embedSWF()` method.

The `embedSWF()` method has the following signature:

```
embedSWF(
    swfUrlStr:String,
    replaceElemIdStr:String,
    heightStr:String,
    widthStr:String,
    swfVersionStr:String,
    xiSwfUrlStr:String,
    flashvars:Object,
    params:Object,
    attributes:Object
)
```

The following is an example of the `embedSWF()` method in an HTML wrapper:

```
swfobject.embedSWF("TestProject.swf", "flashContent","100%", "100%", "10.0.0",
"playerProductInstall.swf", flashvars, params, attributes);
```

You set the values of several of these arguments in the HTML wrapper's script prior to passing them to the `embedSWF()` method. The following table describes these arguments.

Argument	Description
`swfUrlStr`	Defines the location of the application built with Flex. In most cases, this is the name of the output SWF file. If you use Flash Builder to generate a wrapper, the default value is "*project_name*.swf".
`replaceElemIdStr`	The name of the alternative content that appears if Flash Player is not available. You define the alternative content in a `<div>` tag. For an example, view the source of HTML wrapper that is generated by Flash Builder.
`heightStr`	The height of the application built with Flex. If you use Flash Builder to generate a wrapper, this argument is the value of the {`height`} token.
`widthStr`	The width of the application built with Flex. If you use Flash Builder to generate a wrapper, this argument is the value of the {`width`} token.
`swfVersionStr`	The minimum version of Flash Player that is required to run the application built with Flex. The default value is "10.0.0". Set this property to "0" to disable version detection. If you use Flash Builder to generate a wrapper, this value is made up of the {`version_major`}, {`version_minor`}, and {`version_revision`} tokens.
`xiSwfUrlStr`	Enables Express Install. Set this argument to the location of the playerProductInstall.swf file. The default value is "playerProductInstall.swf". This SWF file is in the same directory as the HTML wrapper. If you deploy it to another location, change the value of the `xiSwfUrlStr` argument to the new path. To disable Express Install, set the value of this argument to an empty string. If you use Flash Builder to generate a wrapper, this argument is the value of the {`expressInstallSwf`} token.

Argument	Description

`flashvars`	Adds `flashVars` variables to your template. To do this, attach dynamic properties to the flashvars object in the HTML template.
	The following example adds `firstName` and `lastName` as dynamic properties to flashvars object:
	`var flashvars = {};`
	`flashvars.firstName = "Nick";`
	`flashvars.lastName = "Danger";`
	For more information about using `flashVars` variables in applications built with Flex, see “Passing request data with flashVars properties” on page 229.

`params`	Sets parameters for the SWF object. These properties typically define how it interacts with the HTML wrapper or appears in the browser.
	You can set the values of the following properties by using the `params` argument:
	• `menu`
	• `quality`
	• `scale`
	• `salign`
	• `wmode`
	• `bgcolor`
	• `base`
	• `flashvars`
	• `devicefont`
	• `allowscriptaccess`
	• `seamlesstabbing`
	• `allowfullscreen`
	• `allownetworking`
	For more information about these properties, see the property’s description in “About the object and embed tags” on page 2564.
	The following example adds several parameters to the `params` object in the HTML wrapper:
	`var params = {};`
	`params.quality = "high";`
	`params.bgcolor = "${bgcolor}";`
	`params.allowscriptaccess = "sameDomain";`

Argument	Description

`attributes`	Sets attributes for the SWF object.
	You can set the values of the following properties with the `attributes` argument:
	• `id`
	• `name`
	• `class`
	• `align`
	The `id` and `name` properties are required.
	For more information about these properties, see the property's description in "About the object and embed tags" on page 2564.
	The following example adds the `id`, `name`, and `align` properties to the attributes object.
	`var attributes = {};`
	`attributes.id = "${application}";`
	`attributes.name = "${application}";`
	`attributes.align = "middle";`

For more information on SWFObject 2, see http://code.google.com/p/swfobject/wiki/documentation.

Using deep linking in the wrapper

Deep linking lets users navigate the history of their interactions within the application by using the browser's Forward and Back buttons. Deep linking also lets users read and write to the browser's address bar.

To enable or disable deep linking in the Flash Builder wrapper:

1. Select Project > Properties.
2. Select Flex Compiler from the list on the left of the Project Properties dialog box.
3. Select the Enable Integration With Browser Navigation option to enable deep linking. Deselect this option to disable deep linking.
4. Click OK to save your changes.

When you deploy an application that uses deep linking, you must deploy the files in the /templates/swfobject/history directory with that application.

Using Express Install in the wrapper

The recommended method of ensuring that Flash Player can run the application on the client is to use Express Install. With Express Install, you can detect when users do not have the latest version of Flash Player, and you can initiate an update process that installs the latest version of the player. The updated Player can be installed from the Adobe website or from a local intranet site. When the installation is complete, users are directed back your website, where they can run your application.

Express Install runs a SWF file in the existing Flash Player to upgrade users to the latest version of the player. As a result, Express Install requires that Flash Player already be installed on the client, and that it be version 6.0.65 or later. The Express Install feature also relies on JavaScript detection logic in the browser to ensure that the player required to start the process exists. As a result, the browser must have JavaScript enabled for Express Install to work. If the player on the client is not new enough to support Express Install, you can display alternate content, redirect the user to the Flash Player download page, or initiate another type of Flash Player upgrade experience.

By default, Express Install is enabled in the wrapper generated by Flash Builder.

To enable or disable Express Install in Flash Builder:

1. Select Project > Properties.
2. Select Flex Compiler from the list on the left of the Project Properties dialog box.
3. Select the Use Express Install option to enable Express Install. Deselect the option to disable Express Install.
4. Click OK to save your changes.

When you deploy an application with a wrapper that supports Express Install, you must also deploy the playerProductInstall.swf file. This file is located in the /templates/swfobject directory.

Create a custom wrapper

You can write your own wrapper for your SWF files rather than use the wrapper generated by Flash Builder. Your own wrapper can be simple HTML, or it can be a JavaServer Page (JSP), a PHP page, an Active Server Page (ASP), or anything that can return HTML that is rendered in your client's browser. Typically, you integrate wrapper logic into your website's own HTML templates.

You can create a simple wrapper that uses SWFObject 2 but not other features such as deep linking and Express Install. You do this by creating an HTML file that imports the swfobject.js file. You then call the `swfobject.embedSWF()` method. The following is a simple wrapper that uses SWFObject 2:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <title>wrapper/SimpleSwfObjectWrapper.html</title>
        <script type="text/javascript" src="swfobject.js"></script>
        <script type="text/javascript">
            var swfVersionStr = "0";
            var xiSwfUrlStr = "";
            var flashvars = {};
            var params = {};
            params.quality = "high";
            params.bgcolor = "#ffffff";
            params.allowscriptaccess = "sameDomain";
            var attributes = {};
            attributes.id = "TestProject";
            attributes.name = "TestProject";
            attributes.align = "middle";
            swfobject.embedSWF(
                "TestProject.swf", "flashContent",
                "100%", "100%",
                swfVersionStr, xiSwfUrlStr,
                flashvars, params, attributes);
        </script>
    </head>
    <body>
        <div id="flashContent"/>
   </body>
</html>
```

This wrapper sets the `swfVersionStr` property to 0 and the `xiSwfUrlStr` property to an empty string. Doing this disables Express Install and deep linking functionality. These features improve the user experience and should be omitted only after careful consideration.

When you deploy this example wrapper, you must also deploy the swfobject.js file from the /templates/swfobject directory.

To write an even simpler wrapper, you can use just the `<object>` and `<embed>` tags rather than the SWFObject 2 JavaScript code.

A basic wrapper consists of the following files:

HTML page This is the file that the client browser requests directly. It typically defines two possible experiences (one for users with JavaScript enabled and one for users without JavaScript enabled). This page also references a separate JavaScript file in a `<script>` tag.

JavaScript file The JavaScript file referenced by the `<script>` tag in the HTML page defines the following:

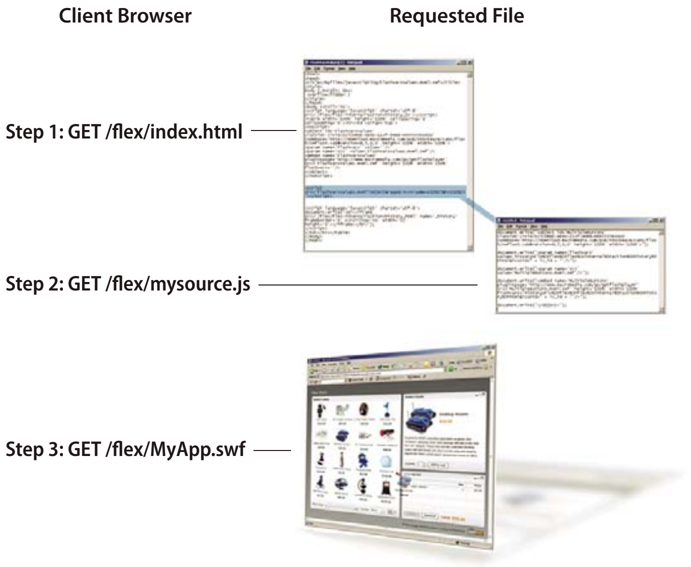
- `<object>` tag This tag embeds the SWF file for Internet Explorer.
- `<embed>` tag This tag embeds the SWF file for Netscape-based browsers.

In the default HTML template, the JavaScript file is named swfobject.js. This JavaScript file contains the SWFObject 2 logic. If you want to create your own basic wrapper, you can load a different JavaScript file that only has `<object>` and `<embed>` tags in it.

The client first requests the HTML page. If the user's browser has JavaScript enabled, the HTML page then loads the JavaScript file. The JavaScript file embeds the application's SWF file.

To make your application respond immediately without user interaction, use a `<script>` tag to load the JavaScript file that contains the embedding logic. Do not write the embed logic directly in the HTML file. Controls that are directly loaded by the embedding page require activation before they will run in Microsoft Internet Explorer 6 or later. If you load the controls directly in the HTML page, users will be required to activate those controls before they can use them by clicking on the control or giving it focus. This is undesirable because you want your application to run immediately when the page loads, not after the user interacts with the control.

The following example illustrates a typical series of requests that a JavaScript-enabled client browser makes when requesting a simple HTML wrapper:


Client Browser
Requested File
Step 1: GET /flex/index.html
Step 2: GET /flex/mysource.js
Step 3: GET /flex/MyApp.swf

The following example shows a simple HTML page and JavaScript file that embeds an application named MyApp:

```
<!-- index.html -->
<!-- saved from url=(0014)about:internet -->
<html>
    <body>
        <script src="mysource.js"></script>
    </body>
</html>

<!-- mysource.js -->
document.write("<object id='MyApp' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
height='100%' width='100%'>");
document.write("<param name='movie' value='MyApp.swf'/>");
document.write("<embed name='MyApp' src='MyApp.swf' height='100%' width='100%'/>");
document.write("</object>");
```

The first line in the index.html page is called the Mark of the Web (MOTW). Adding the MOTW to your wrapper is optional. However, if you do not add the MOTW to your wrapper, your application might not open in the expected security zone within Internet Explorer. The following example MOTW forces Internet Explorer to open the page in the Internet zone:

```
<!-- saved from url=(0014)about:internet -->
```

In general, add a MOTW when you are previewing pages locally before publishing them on a server. For more information about the MOTW, see http://msdn.microsoft.com/en-us/library/ms537628%28VS.85%29.aspx.

To support browsers that do not have scripting enabled, you can add a `<noscript>` block to your simple wrapper. The tags in this block of code usually mirror the output of the `document.write()` methods in the embedded JavaScript file. The following example adds the `<noscript>` block:

```
<!-- index.html -->
<!-- saved from url=(0014)about:internet -->
<html>
    <body>
        <script src="mysource.js"></script>
        <noscript>
            <object id='MyApp' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
                height='100%' width='100%'>
                <param name='src' value='MyApp.swf'/>
                <embed name='MultipleButtons' src='MyApp.swf' height='100%'
                    width='100%'/>
            </object>
        </noscript>
    </body>
</html>
```

This simple wrapper does not include any assurances that the client can run the application. If they do not have a minimum version of Flash Player, their attempt to run this application will fail and they might be prompted to upgrade. The upgrade experience is considerably better with Express Install. If the user's Player does not meet the minimum requirements, the new player is automatically installed for them. You add Express Install by editing your wrapper. For more information, see "Adding Express Install to a custom wrapper" on page 2563.

With a generic wrapper, a user who clicks the Back and Forward buttons in their browser navigates the HTML pages in the browser's history and not the history of their interactions within the application. Deep linking lets users navigate their interactions with the application by using the Back and Forward buttons in their browser. You can add deep linking by editing your wrapper and deploying additional files with it. For more information, see "Adding deep linking to a custom wrapper" on page 2563.

For more information about using the `<object>` and `<embed>` tags, see "About the object and embed tags" on page 2564.

About customizing a wrapper

Use the following guidelines when you create a custom wrapper.

- You can use the HTML template in the /templates/swfobject directory as a guide to adding new features to the wrapper. For information about the template, see "About the HTML template" on page 2553.
- You must embed the SWF file and not the MXML file of an application built with Flex. Set the value of the `src` property to *mxml_filename*`.swf`.

  The following example defines the `src` property of the `<object>` tag for an MXML application called MyApp.mxml:

  ```
  <param name='src' value='MyApp.swf'>
  ```

  The `<embed>` tag uses the `src` property to define the source of the SWF file:

  ```
  src='MyApp.mxml.swf'
  ```

- Do not include periods or other special characters in the `id` and `name` properties of the `<object>` and `<embed>` tags. These tags identify the SWF object on the page, and you use them when you use the ExternalInterface API. This API lets Flex communicate with the wrapper, and vice versa. For more information about using the ExternalInterface API, see "Communicating with the wrapper" on page 226.

- Do not put the contents of the JavaScript file directly in the HTML page. This causes Internet Explorer to prompt the user before enabling Flash Player. If the client has "Disable Script Debugging (Internet Explorer)" unchecked in Internet Explorer's advanced settings, the browser still prompts the user to load the ActiveX plug-in before running it.
- If you use both the `<object>` and the `<embed>` tags in your custom wrapper, use identical values for each attribute to ensure consistent playback across browsers. For more information about the `<object>` and the `<embed>` tags, see "About the object and embed tags" on page 2564.
- To add support for deep linking, follow the instructions in "Adding deep linking to a custom wrapper" on page 2563.
- To add support for Flash Player detection, follow the instructions in "Adding Express Install to a custom wrapper" on page 2563.
- When using Flash Builder, the default wrapper includes deep linking and Express Install support. You can disable one or both of these features by using the Compiler Properties dialog box. You can also use this dialog box to set the minimum required version of Flash Player.

Adding Express Install to a custom wrapper

Express Install uses the existing Flash Player to upgrade users to the latest version of the Player when it is required for viewing content. Express Install requires that the user has Flash Player 6.0.65 or later installed on a Mac or Windows operating system. Express Install also requires the user to have JavaScript enabled in the browser.

Support for Express Install is included in the wrapper that is generated by Flash Builder. You can enable or disable it in Flash Builder prior to generating the wrapper in Flash Builder's Flex Compiler Properties dialog box.

If you write your own custom wrapper and want to use Express Install, you must add support for it manually. Adobe recommends that you use the Express Install functionality in the wrapper template at /templates/swfobject directory as a base.

Adding Express Install support from the wrapper template involves adding JavaScript to your main wrapper file, as well as deploying the swfobject.js file. In addition, you must deploy the playerProductInstall.swf file with your application. This file is located in the SWF file's output directory when you build a project in Flash Builder, or the /templates/swfobject directory if you use the SDK.

You can also use the Flash Player Detection Kit to add Express Install support to your custom wrapper.

More Help topics

Flash Player Detection Kit

Adding deep linking to a custom wrapper

Support for deep linking, also known as history management, is included by default in the HTML that is wrapper generated by Flash Builder. If you write your own wrapper, however, you must add it manually or use the HTML template file as a base.

To add deep linking support, you reference the history.js and history.css files in your wrapper. The following sample wrapper references these files in the /history subdirectory:

```
<!-- index.html -->
<!-- saved from url=(0014)about:internet -->
<html>
<body>
    <link rel="stylesheet" type="text/css" href="history/history.css" />
    <script type="text/javascript" src="history/history.js"></script>
    <script src="mysource.js"></script>
</body>
</html>
```

The history.js file records actions for deep linking. The history.css file defines styles for the history files. To support deep linking in your application built with Flex, you must deploy the following files with your wrapper:

- history.js
- historyFrame.html
- history.css

These files are generated by Flash Builder in the /history directory. If you are using the SDK, you can get these files from the /templates/swfobject/history directory.

If you do not add deep linking support to your wrapper, you cannot use the BrowserManager class in applications built with Flex.

More Help topics

“Deep linking” on page 2022

About the object and embed tags

The `<object>` and `<embed>` tags embed SWF files in HTML. The wrapper generated by Flash Builder hides the details of these tags from you for most common use cases.

Adobe recommends using SWFObject 2 rather than the `<object>` and `<embed>` tags to embed your SWF file, but these tags are useful in some cases, such as if you want to create a custom wrapper or if there is a likelihood that your users will have JavaScript disabled in their browsers.

For browsers with scripting disabled, the default HTML template uses the `<object>` and `<embed>` tags to embed your application. These tags are children of the `<noscript>` tag, which means that the browser executes them when scripting is disabled on the client. When you compile a project in Flash Builder, Flex Builder sets the values of the tokens within these tags for you. When you compile an application with the SDK, you must manually edit the template and set the values of these tags.

You can also use the `<object>` and `<embed>` tags tags in simple HTML wrappers. For more complex wrappers, or wrappers that support features such as Express Install, you should use the SWFObject 2 functionality as described in “About SWFObject 2” on page 2555.

The `<object>` and `<embed>` tags support a set of properties that add additional functionality to the wrapper. These properties let you change the appearance of the SWF file on the page or change some of its properties such as the title or language. If you want to customize your wrapper, you can add these properties to the wrapper.

The `<object>` tag is supported by Internet Explorer 3.0 or later on Windows platforms or any browser that supports the use of the Flash ActiveX control. The `<embed>` tag is supported by Netscape Navigator 2.0 or later, or browsers that support the use of the Netscape-compatible plug-in version of Flash Player.

When an ActiveX-enabled browser loads the HTML page, it reads the values set on the `<object>` and ignores the `<embed>` tag. When browsers using the Flash plug-in load the HTML page, they read the values set on the `<embed>` tag and ignore the `<object>` tag. Make sure that the properties for each tag are identical, unless you want different results depending on the user's browser.

The following are required attributes of the `<object>` tag:

- `height`
- `width`
- `classid`

All other properties are optional and you set their values in separate, named `<param>` tags.

Although the `movie` property is technically an optional tag, without it, there is no reference to the SWF file you want the client to load. Therefore, your wrapper should always set the `movie` parameter in the `<object>` tag.

The following example shows the required properties as attributes of the `<object>` tag, and two optional properties, `movie` and `quality`, as `<param>` child tags:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="100" height="100">
    <param name="movie" value="moviename.swf">
    <param name="quality" value="high">
</object>
```

For the `<embed>` tag, all settings are attributes that appear between the angle brackets of the opening `<embed>` tag. The `<embed>` tag requires the `height` and `width` attributes. The `<embed>` tag does not require a `classid` attribute. As with the `movie` parameter of the `<object>` tag, the `src` attribute of the `<embed>` tag provides the reference to the application. Without it, there would be no SWF file, so you should consider it a required attribute.

The following example shows a simple `<embed>` tag with the optional `quality` attribute:

```
<embed src="moviename.swf" width="100" height="100" quality="high"></embed>
```

To use both tags together, position the `<embed>` tag just before the closing `</object>` tag, as the following example shows:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="100" height="100">
    <param name="movie" value="moviename.swf">
    <param name="quality" value="high">
    <embed src="moviename.swf" width="100" height="100" quality="high"></embed>
</object>
```

When you define parameters for the `<object>` tag, also add them as tag properties to the `<embed>` tag so that the SWF file appears the same on the page regardless of the client's browser.

If you are using the included HTML wrapper template, you will need to replace tokens such as `${height}` and `${width}` with absolute values before deploying the wrapper. For more information, see "About the HTML template" on page 2553.

Not all properties are supported by both the `<object>` and the `<embed>` tags. For example, the `id` property is used only by the `<object>` tag, just as the `name` property is used only by the `<embed>` tag.

In some cases, the `<object>` and `<embed>` tag properties duplicate properties that you can set on the `<s:Application>` tag in the application source code. For example, you can set the `height` and `width` properties of the SWF file on the `<object>` and `<embed>` tags or you can set them on the `<s:Application>` tag.

The following table describes the supported `<object>` and `<embed>` tag properties:

Property	Type	Description
`align`	String	Specifies the position of the SWF file. The `align` property supports the following values: • `bottom`: Vertically aligns the bottom of the SWF file with the current baseline. This is typically the default value. • `middle`: Vertically aligns the middle of the SWF file with the current baseline. • `top`: Vertically aligns the top of the SWF file with the top of the current text line. • `left`: Horizontally aligns the SWF file to the left margin. • `right`: Horizontally aligns the SWF file to the right margin.
`allowNetworking`	String	Restricts communication between the HTML page and the SWF file. This property affects more APIs than the `allowScriptAccess` property. The `allowNetworking` property supports the following values: • `all`: No networking restrictions. Flash Player behaves normally. This is typically the default. • `internal`: SWF files cannot call browser navigation or browser interaction APIs (such as the `ExternalInterface.call()`, `fscommand()`, and `navigateToURL()` methods), but can call other networking APIs. • `none`: SWF files cannot call networking or SWF-to-SWF file communication APIs. In addition to the APIs restricted by the `internal` value, these include other methods such as `URLLoader.load()`, `Security.loadPolicyFile()`, and `SharedObject.getLocal()`. For more information, see ActionScript 3.0 Developer's Guide.
`allowScriptAccess`	String	Controls the ability to perform outbound scripting from within the SWF file. The `allowScriptAccess` property can prevent a SWF file hosted from one domain from accessing a script in an HTML page that comes from another domain. Valid values are as follows: • `always`: Outbound scripting always succeeds. • `samedomain:` Outbound scripting succeds only if the application is from the same domain as the HTML page. • `never`: Outbound scripting always fails. Using this value is deprecated and not recommended, and should not be necessary if you do not serve untrusted SWF files from your own domain. If you do need to serve untrusted SWF files, Adobe recommends that you create a distinct subdomain and place all untrusted content there. This property affects the following operations: • `ExternalInterface.call()` • `fscommand()` • `navigateToURL()`, when used with `javascript` or another scripting scheme • `navigateToURL()`, when used with window name of `_self`, `_parent`, or `_top`. For more information, see ActionScript 3.0 Developer's Guide.
`archive`	String	Specifies a space-separated list of URIs for archives containing resources used by the application, which may include the resources specified by the `classid` and `data` properties. Preloading archives can result in reduced load times for applications. Archives specified as relative URIs are interpreted relative to the `codebase` property.

Property	Type	Description
base	String	Specifies the base directory or URL used to resolve relative path statements in ActionScript.
bgcolor	String	Specifies the background color of the application.
border	int	Specifies the width of the SWF file’s border, in pixels. The default value for this property depends on the user agent.
classid	String	Defines the classid of Flash Player. This identifies the ActiveX control for the browser. Internet Explorer 3.0 or later on Windows 9x, Windows 2000, Windows NT, Windows ME, and Windows XP prompt the user with a dialog box asking if they would like to auto-install Flash Player if it's not already installed. This process can occur without the user having to restart the browser. This property is used for the <object> tag only. For the <object> tag, you set the value of this property as an attribute of the <object> tag and not as a <param> tag.
codetype	String	Defines the content type of data expected when downloading the application specified by the classid property The codetype property is optional but recommended when the classid property is specified; it lets the browser avoid loading unsupported content types. The default value of the codetype property is the value of the type property.
data	String	Specifies the location of the application’s data; for example, instance image data for objects that define images.
declare	Boolean	Makes the current SWF file’s definition a declaration only. The SWF file must be instantiated by a subsequent object definition referring to this declaration.
devicefont	Boolean	Specifies whether static text objects for which the deviceFont option is not selected are drawn using a device font anyway, if the needed fonts are available from the operating system.
dir	String	Specifies the base direction of text in an element’s content and attribute values. It also specifies the directionality of tables. Valid values are LTR (left-to-right text or table) and RTL (right-to-left text or table).
flashVars	String	Sends variables to the application. The format is a set of name-value pairs, each separated by an ampersand (&). Most browsers support string sizes of up to 64 KB (65535 bytes) in length. The default value of this property is typically an empty string. The value of this property is URL encoded prior to being sent to the application. For more information on using flashVars to pass variables to applications built with Flex, see “Communicating with the wrapper” on page 226.
height	int	The height of the application SWF file. For the <object> tag, you set the value of this property as an attribute of the <object> tag and not as a <param> child tag.
hspace	int	Specifies the amount of white space inserted to the left and right of the SWF file. The default value is typically not specified, but is generally a small, nonzero length.
id	String	Identifies the SWF file to the host environment (a web browser, for example) so that it can be referenced by using a scripting language such as VBScript or JavaScript. The id property is only used with the <object> tag. It is equivalent to the name property used with the <embed> tag.
lang	String	Specifies the base language of an element’s property values and text content. The default value is typically unknown. The browser can use language information specified using the lang property to control rendering in a variety of ways.

Property	Type	Description
menu	Boolean	Changes the appearance of the menu that appears when users right-click over an application in Flash Player. Set to true to display the entire menu. Set to false to display only the About and Settings options on the menu. The default value is typically true.
movie	String	Specifies the location of the SWF file. The movie property is only used with the <object> tag. It is equivalent to the src property used with the <embed> tag.
name	String	Identifies the SWF file to the host environment (a web browser, typically) so that it can be referenced by using a scripting language. The name property is only used with the <embed> tag. It is equivalent to the id property used with the <object> tag.
quality	String	Defines the quality of playback in Flash Player. Valid values of quality are low, medium, high, autolow, autohigh, and best. The default value is typically best. The low setting favors playback speed over appearance and never uses anti-aliasing. The autolow setting emphasizes speed at first but improves appearance whenever possible. Playback begins with anti-aliasing turned off. If Flash Player detects that the processor can handle it, anti-aliasing is turned on. The autohigh setting emphasizes playback speed and appearance equally at first, but sacrifices appearance for playback speed if necessary. Playback begins with anti-aliasing turned on. If the actual frame rate drops below the specified frame rate, anti-aliasing is turned off to improve playback speed. Use this setting to emulate the View > Antialias setting in Flash. The medium setting applies some anti-aliasing and does not smooth bitmaps. The high setting favors appearance over playback speed and always applies anti-aliasing. The best setting provides the best display quality and does not consider playback speed. All output is anti-aliased and all bitmaps are smoothed.
salign	String	Positions the SWF file within the browser. Valid values are L, T, R, B, TL, TR, BL, and BR. L, R, T, and B align the SWF file along the left, right, top, or bottom edge, respectively, of the browser window and crop the remaining three sides as needed. TL and TR align the SWF file to the top-left and top-right corner, respectively, of the browser window and crop the bottom and remaining right or left side as needed. BL and BR align the SWF file to the bottom-left and bottom-right corner, respectively, of the browser window and crop the top and remaining right or left side as needed.
scale	String	Defines how the browser fills the screen with the SWF file. The default value is typically showall. Valid values of the scale property are showall, noborder, and exactfit. Set to showall to make the entire SWF file visible in the specified area without distortion, while maintaining the original aspect ratio of the SWF file. Borders may appear on two sides of the SWF file. Set to noborder to scale the SWF file to fill the specified area, without distortion but possibly with some cropping, while maintaining the original aspect ratio of the SWF file. Set to exactfit to make the entire SWF file visible in the specified area without trying to preserve the original aspect ratio. Distortion may occur.
src	String	Specifies the location of the SWF file. The src property is only used with the <embed> tag. It is equivalent to the movie property used with the <object> tag.

Property	Type	Description
standby	String	Defines a message that the browser displays while loading the object's implementation and data.
style	String	Specifies style information for the SWF file. The syntax of the value of the style property is determined by the default style sheet language. In CSS, property declarations have the form "name:value" and are separated by a semicolon. Styles set with this property do not affect components or the Application container in the application. Rather, they apply to the SWF file as it appears on the HTML page.
supportembed	Boolean	Determines whether the Netscape-specific <embed> tag is supported. The supportembed property is optional, and the default value is typically true. Set to false to prevent the <embed> tag from being read by the browser.
tabindex	int	Specifies the position of the SWF file in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros.
title	String	Displays information about the SWF file. Values of the title property can be rendered by browsers or other user agents in different ways. For example, some browsers display the title as a ToolTip. Audio user agents might speak the title information in a similar context.
type	String	Specifies the content type for the data specified by the data property. The type property is optional but recommended when data is specified; it prevents the browser from loading unsupported content types. If the value of this property differs from the HTTP Content-Type returned by the server, the HTTP Content-Type takes precedence.
usemap	String	Associates an image map with the SWF file. The image map is defined by a map element. The value of usemap must match the value of the name attribute of the associated map element.
vspace	int	Specifies the amount of white space inserted above and below the SWF file. The default value is typically not specified, but is generally a small, nonzero length.
width	int	The width of the application SWF file. For the <object> tag, you set the value of this property as an attribute of the <object> tag and not as a <param> tag.
wmode	String	Sets the Window Mode property of the SWF file for transparency, layering, and positioning in the browser. Valid values of wmode are window, opaque, and transparent. Set to window to play the SWF in its own rectangular window on a web page. Set to opaque to hide everything on the page behind it. Set to transparent so that the background of the HTML page shows through all transparent portions of the SWF file. This can slow animation performance. To make sections of your SWF file transparent, you must set the alpha property to 0. To make your application's background transparent, set the backgroundAlpha property on the <fx:Application> tag to 0. The wmode property is not supported in all browsers and platforms.

The <object> and <embed> tags can also take additional properties that are not supported by applications built with Flex. These unsupported properties are listed in "Unsupported Flash Player properties" on page 2570.

Unsupported Flash Player properties

Some optional Flash Player properties do not apply to applications built with Flex. These are properties that involve movie frames and looping. The following properties have no effect when used with Flex:

- `loop`
- `play`