Learning ACTIONSCRIPT® 3.0

Legal notices

Contents

Chapter 1: Introduction to ActionScript 3.0

About ActionScript

ActionScript is the programming language for the Adobe® Flash® Player and Adobe® AIR™ run-time environments. It enables interactivity, data handling, and much more in Flash, Flex, and AIR content and applications.

ActionScript executes in the ActionScript Virtual Machine (AVM), which is part of Flash Player and AIR. ActionScript code is typically transformed into bytecode format by a compiler. (*Bytecode* is a type of programming language that's written and understood by computers.) Examples of compilers include the one built in to Adobe® Flash® Professional and the one that is built in to Adobe® Flash® Builder™ and available in the Adobe® Flex™ SDK. The bytecode is embedded in SWF files, which Flash Player and AIR execute.

ActionScript 3.0 offers a robust programming model that is familiar to developers with a basic knowledge of object-oriented programming. Some of the key features of ActionScript 3.0 that improve over previous ActionScript versions include the following:

- A new ActionScript Virtual Machine, called AVM2, that uses a new bytecode instruction set and provides significant performance improvements
- A more modern compiler code base that performs deeper optimizations than previous versions of the compiler
- An expanded and improved application programming interface (API), with low-level control of objects and a true object-oriented model
- An XML API based on the ECMAScript for XML (E4X) specification (ECMA-357 edition 2). E4X is a language extension to ECMAScript that adds XML as a native data type of the language.
- An event model based on the Document Object Model (DOM) Level 3 Events Specification

Advantages of ActionScript 3.0

ActionScript 3.0 goes beyond the scripting capabilities of previous versions of ActionScript. It is designed to facilitate the creation of highly complex applications with large data sets and object-oriented, reusable code bases. ActionScript 3.0 is not required for content that runs in Adobe Flash Player. However, it opens the door to performance improvements that are only available with the AVM2 (the ActionScript 3.0 virtual machine). ActionScript 3.0 code can execute up to ten times faster than legacy ActionScript code.

The previous version of ActionScript Virtual Machine, AVM1, executes ActionScript 1.0 and ActionScript 2.0 code. Flash Player 9 and 10 support AVM1 for backward compatibility.

What's new in ActionScript 3.0

ActionScript 3.0 contains many classes and features that are similar to ActionScript 1.0 and 2.0. However, ActionScript 3.0 is architecturally and conceptually different from previous versions of ActionScript. The enhancements in ActionScript 3.0 include new features of the core language and an improved API that provides increased control of low-level objects.

Core language features

The core language defines the basic building blocks of the programming language, such as statements, expressions, conditions, loops, and types. ActionScript 3.0 contains many features that speed up the development process.

Run-time exceptions

ActionScript 3.0 reports more error conditions than previous versions of ActionScript. Run-time exceptions are used for common error conditions, improving the debugging experience and enabling you to develop applications that handle errors robustly. Run-time errors can provide stack traces annotated with source file and line number information, helping you quickly pinpoint errors.

Run-time types

In ActionScript 3.0, type information is preserved at run time. This information is used to perform run-time type checking, improving the system's type safety. Type information is also used to represent variables in native machine representations, which improves performance and reduces memory usage. By way of comparison, in ActionScript 2.0 type annotations are primarily a developer aid and all values are dynamically typed at run time.

Sealed classes

ActionScript 3.0 includes the concept of sealed classes. A sealed class possesses only the fixed set of properties and methods that are defined at compile time; additional properties and methods cannot be added. The inability of changing a class at run time enables stricter compile-time checking, resulting in more robust programs. It also improves memory usage by not requiring an internal hash table for each object instance. Dynamic classes are also possible using the `dynamic` keyword. All classes in ActionScript 3.0 are sealed by default, but can be declared to be dynamic with the `dynamic` keyword.

Method closures

ActionScript 3.0 enables a method closure to automatically remember its original object instance. This feature is useful for event handling. In ActionScript 2.0, method closures do not remember what object instance they were extracted from, leading to unexpected behavior when the method closure is called.

ECMAScript for XML (E4X)

ActionScript 3.0 implements ECMAScript for XML (E4X), recently standardized as ECMA-357. E4X offers a natural, fluent set of language constructs for manipulating XML. In contrast to traditional XML-parsing APIs, XML with E4X performs like a native data type of the language. E4X streamlines the development of applications that manipulate XML by drastically reducing the amount of code needed.

To view the ECMA E4X specification, go to www.ecma-international.org.

Regular expressions

ActionScript 3.0 includes native support for regular expressions so that you can quickly search for and manipulate strings. ActionScript 3.0 implements support for regular expressions as they are defined in the ECMAScript (ECMA-262) edition 3 language specification.

Namespaces

Namespaces are similar to the traditional access specifiers used to control visibility of declarations (`public`, `private`, `protected`). They work as custom access specifiers, which can have names of your choice. Namespaces are outfitted with a Universal Resource Identifier (URI) to avoid collisions, and are also used to represent XML namespaces when you work with E4X.

New primitive types

ActionScript 3.0 contains three numeric types: Number, int, and uint. Number represents a double-precision, floating-point number. The int type is a 32-bit signed integer that lets ActionScript code take advantage of the fast integer math capabilities of the CPU. The int type is useful for loop counters and variables where integers are used. The uint type is an unsigned, 32-bit integer type that is useful for RGB color values, byte counts, and more. In contrast, ActionScript 2.0 only has a single numeric type, Number.

API features

The APIs in ActionScript 3.0 contain many classes that allow you to control objects at a low level. The architecture of the language is designed to be more intuitive than previous versions. While there are too many classes to cover in detail, some significant differences are worth noting.

DOM3 event model

Document Object Model Level 3 event model (DOM3) provides a standard way of generating and handling event messages. This event model is designed to allow objects within applications to interact and communicate, maintain their state, and respond to change. The ActionScript 3.0 event model is patterned after the World Wide Web Consortium DOM Level 3 Events Specification. This model provides a clearer and more efficient mechanism than the event systems available in previous versions of ActionScript.

Events and error events are located in the flash.events package. The Flash Professional components and Flex framework use the same event model, so the event system is unified across the Flash Platform.

Display list API

The API for accessing the display list—the tree that contains any visual elements in the application—consists of classes for working with visual primitives.

The Sprite class is a lightweight building block, designed to be a base class for visual elements such as user interface components. The Shape class represents raw vector shapes. These classes can be instantiated naturally with the `new` operator and can be dynamically reparented at any time.

Depth management is automatic. Methods are provided for specifying and managing the stacking order of objects.

Handling dynamic data and content

ActionScript 3.0 contains mechanisms for loading and handling assets and data in your application that are intuitive and consistent across the API. The Loader class provides a single mechanism for loading SWF files and image assets and provides a way to access detailed information about loaded content. The URLLoaderclass provides a separate mechanism for loading text and binary data in data-driven applications. The Socket class provides a means to read and write binary data to server sockets in any format.

Low-level data access

Various APIs provide low-level access to data. For data that is being downloaded, the URLStream class provides access to data as raw binary data while it is being downloaded. The ByteArray class lets you optimize reading, writing, and working with binary data. The sound API provides detailed control of sound through the SoundChannel and SoundMixer classes. Security APIs provide information about the security privileges of a SWF file or loaded content, enabling you to handle security errors.

Working with text

ActionScript 3.0 contains a flash.text package for all text-related APIs. The TextLineMetrics class provides detailed metrics for a line of text within a text field; it replaces the `TextFormat.getTextExtent()` method in ActionScript 2.0. The TextField class contains low-level methods that provide specific information about a line of text or a single character in a text field. For example, the `getCharBoundaries()` method returns a rectangle representing the bounding box of a character. The `getCharIndexAtPoint()` method returns the index of the character at a specified point. The `getFirstCharInParagraph()` method returns the index of the first character in a paragraph. Line-level methods include `getLineLength()`, which returns the number of characters in a specified line of text, and `getLineText()`, which returns the text of the specified line. The Font class provides a means to manage embedded fonts in SWF files.

For even lower-level control over text, the classes in the flash.text.engine package make up the Flash Text Engine. This set of classes provide low-level control over text and are designed for creating text frameworks and components.

Chapter 2: Getting started with ActionScript

Programming fundamentals

Since ActionScript is a programming language, it can help you to learn ActionScript if you first understand a few general computer programming concepts.

What computer programs do

First of all, it's useful to have a conceptual idea of what a computer program is and what it does. There are two main aspects to a computer program:

- A program is a series of instructions or steps for the computer to carry out.
- Each step ultimately involves manipulating some piece of information or data.

In a general sense, a computer program is just a list of step-by-step instructions that you give to the computer, which it performs one by one. Each individual instruction is known as a *statement*. In ActionScript, each statement is written with a semicolon at the end.

In essence, all that a given instruction in a program does is manipulate some bit of data that's stored in the computer's memory. A simple example is instructing the computer to add two numbers and store the result in its memory. A more complex example is if there is a rectangle drawn on the screen and you want to write a program to move it somewhere else on the screen. The computer remembers certain information about the rectangle: the x, y coordinates where it's located, how wide and tall it is, what color it is, and so on. Each of those bits of information is stored somewhere in the computer's memory. A program to move the rectangle to a different location would have steps like "change the x coordinate to 200; change the y coordinate to 150." In other words, it would specify new values for the x and y coordinates. Behind the scenes, the computer does something with this data to actually turn those numbers into the image that appears on the computer screen. However, at the basic level of detail it's enough to know that the process of "moving a rectangle on the screen" just involves changing bits of data in the computer's memory.

Variables and constants

Programming mainly involves changing pieces of information in the computer's memory. Consequently, it's important to have a way to represent a single piece of information in a program. A *variable* is a name that represents a value in the computer's memory. As you write statements to manipulate values, you write the variable's name in place of the value. Any time the computer sees the variable name in your program, it looks in its memory and uses the value it finds there. For example, if you have two variables named `value1` and `value2`, each containing a number, to add those two numbers you could write the statement:

```
value1 + value2
```

When it's actually carrying out the steps, the computer looks to see the values in each variable and adds them together.

In ActionScript 3.0, a variable actually consists of three different parts:

- The variable's name
- The type of data that can be stored in the variable
- The actual value stored in the computer's memory

You've seen how the computer uses the name as a placeholder for the value. The data type is also important. When you create a variable in ActionScript, you specify the specific type of data that it is meant to hold. From that point on, your program's instructions can store only that type of data in the variable. You can manipulate the value using the particular characteristics associated with its data type. In ActionScript, to create a variable (known as *declaring* the variable), you use the `var` statement:

```
var value1:Number;
```

This example tells the computer to create a variable named `value1`, which can hold only Number data. ("Number" is a specific data type defined in ActionScript.) You can also store a value in the variable right away:

```
var value2:Number = 17;
```

Adobe Flash Professional

In Flash Professional, there is another way to declare a variable. When you place a movie clip symbol, button symbol, or text field on the Stage, you can give it an instance name in the Property inspector. Behind the scenes, Flash Professional creates a variable with the same name as the instance name. You can use that name in your ActionScript code to represent that Stage item. Suppose, for example, that you have a movie clip symbol on the Stage and you give it the instance name `rocketShip`. Whenever you use the variable `rocketShip` in your ActionScript code, you are in fact manipulating that movie clip.

A *constant* is similar to a variable. It is a name that represents a value in the computer's memory with a specified data type. The difference is that a constant can only be assigned a value one time in the course of an ActionScript application. Once a constant's value is assigned, it is the same throughout the application. The syntax for declaring a constant is almost the same as that for declaring a variable. The only difference is that you use the `const` keyword instead of the `var` keyword:

```
const SALES_TAX_RATE:Number = 0.07;
```

A constant is useful for defining a value that is used in multiple places throughout a project, which don't change under normal circumstances. Using a constant rather than a literal value makes your code more readable. For example, consider two versions of the same code. One multiplies a price by `SALES_TAX_RATE`. The other multiplies the price by `0.07`. The version that uses the `SALES_TAX_RATE` constant is easier to understand. In addition, suppose the value defined by the constant does change. If you use a constant to represent that value throughout your project, you can change the value in one place (the constant declaration). In contrast, you would have to change it in various places if you used hard-coded literal values.

Data types

In ActionScript, there are many data types that you can use as the data type of the variables you create. Some of these data types can be thought of as "simple" or "fundamental" data types:

- String: a textual value, like a name or the text of a book chapter
- Numeric: ActionScript 3.0 includes three specific data types for numeric data:
  - Number: any numeric value, including values with or without a fraction
  - int: an integer (a whole number without a fraction)
  - uint: an "unsigned" integer, meaning a whole number that can't be negative
- Boolean: a true-or-false value, such as whether a switch is on or whether two values are equal

The simple data types represent a single piece of information: for example, a single number or a single sequence of text. However, most of the data types defined in ActionScript could are complex data types. They represent a set of values in a single container. For example, a variable with the data type Date represents a single value (a moment in time). Nevertheless, that date value is represented as several values: the day, month, year, hours, minutes, seconds, and so on, all of which are individual numbers. People generally think of a date as a single value, and you can treat a date as a single value by creating a Date variable. However, internally the computer thinks of it as a group of several values that, put together, define a single date.

Most of the built-in data types, as well as data types defined by programmers, are complex data types. Some of the complex data types you probably recognize are:

- MovieClip: a movie clip symbol
- TextField: a dynamic or input text field
- SimpleButton: a button symbol
- Date: information about a single moment in time (a date and time)

Two words that are often used as synonyms for data type are class and object. A *class* is simply the definition of a data type. It's like a template for all objects of the data type, like saying "all variables of the Example data type have these characteristics: A, B, and C." An *object*, on the other hand, is just an actual instance of a class. For example, a variable whose data type is MovieClip could be described as a MovieClip object. The following are different ways of saying the same thing:

- The data type of the variable `myVariable` is Number.
- The variable `myVariable` is a Number instance.
- The variable `myVariable` is a Number object.
- The variable `myVariable` is an instance of the Number class.

Working with objects

ActionScript is what's known as an object-oriented programming language. Object-oriented programming is simply an approach to programming. It's really nothing more than a way to organize the code in a program, using objects.

Earlier the term "computer program" was defined as a series of steps or instructions that the computer performs. Conceptually, then, you can imagine a computer program as just a single long list of instructions. However, in object-oriented programming, the program instructions are divided among different objects. The code is grouped into chunks of functionality, so related types of functionality or related pieces of information are grouped in one container.

Adobe Flash Professional
If you've worked with symbols in Flash Professional, you're already used to working with objects. Imagine you've defined a movie clip symbol such as a drawing of a rectangle and you've placed a copy of it on the Stage. That movie clip symbol is also (literally) an object in ActionScript; it's an instance of the MovieClip class.

There are various characteristics of the movie clip that you can modify. When it's selected you can change values in the Property inspector like its x coordinate or its width. You can also make various color adjustments like changing its alpha (transparency) or applying a drop-shadow filter to it. Other Flash Professional tools let you make more changes, like using the Free Transform tool to rotate the rectangle. All of these ways that you can modify a movie clip symbol in Flash Professional are also available in ActionScript. You modify the movie clip in ActionScript by changing the pieces of data that are all put together into a single bundle called a MovieClip object.

In ActionScript object-oriented programming, there are three types of characteristics that any class can include:

- Properties
- Methods
- Events

These elements are used to manage the pieces of data used by the program and to decide what actions are carried out and in what order.

Properties

A property represents one of the pieces of data that are bundled together in an object. An example song object can have properties named `artist` and `title`; the MovieClip class has properties like `rotation`, `x`, `width`, and `alpha`. You work with properties like individual variables. In fact, you can think of properties as simply the "child" variables contained in an object.

Here are some examples of ActionScript code that uses properties. This line of code moves the MovieClip named `square` to the x coordinate 100 pixels:

```
square.x = 100;
```

This code uses the rotation property to make the `square` MovieClip rotate to match the rotation of the `triangle` MovieClip:

```
square.rotation = triangle.rotation;
```

This code alters the horizontal scale of the `square` MovieClip making it one-and-a-half times wider than it used to be:

```
square.scaleX = 1.5;
```

Notice the common structure: you use a variable (`square`, `triangle`) as the name of the object, followed by a period (`.`) and then the name of the property (`x`, `rotation`, `scaleX`). The period, known as the *dot operator*, is used to indicate that you're accessing one of the child elements of an object. The whole structure together, "variable name-dot-property name," is used like a single variable, as a name for a single value in the computer's memory.

Methods

A *method* is an action that an object can perform. For example, suppose you've made a movie clip symbol in Flash Professional with several keyframes and animation on its timeline. That movie clip can play, or stop, or be instructed to move the playhead to a particular frame.

This code instructs the MovieClip named `shortFilm` to start playing:

```
shortFilm.play();
```

This line makes the MovieClip named `shortFilm` stop playing (the playhead stops in place, like pausing a video):

```
shortFilm.stop();
```

This code makes a MovieClip named `shortFilm` move its playhead to Frame 1 and stop playing (like rewinding a video):

```
shortFilm.gotoAndStop(1);
```

Methods, like properties, are accessed by writing the object's name (a variable), then a period, and then the name of the method followed by parentheses. The parentheses are the way that you indicate that you are *calling* the method, or in other words, instructing the object to perform that action. Sometimes values (or variables) are placed in the parentheses, as a way to pass along additional information that is necessary to carry out the action. These values are known as method *parameters*. For example, the `gotoAndStop()` method needs information about which frame to go to, so it requires a single parameter in the parentheses. Other methods, like `play()` and `stop()`, are self-explanatory, so they don't require extra information. Nevertheless, they are still written with parentheses.

Unlike properties (and variables), methods aren't used as value placeholders. However, some methods can perform calculations and return a result that can be used like a variable. For example, the Number class's `toString()` method converts the numeric value to its text representation:

```
var numericData:Number = 9;
var textData:String = numericData.toString();
```

For example, you would use the `toString()` method if you wanted to display the value of a Number variable in a text field on the screen. The TextField class's `text` property is defined as a String, so it can contain only text values. (The text `property` represents the actual text content displayed on the screen). This line of code converts the numeric value in the variable `numericData` to text. It then makes the value show up on the screen in the TextField object named `calculatorDisplay`:

```
calculatorDisplay.text = numericData.toString();
```

Events

A computer program is a series of instructions that the computer carries out step-by-step. Some simple computer programs consist of nothing more than a few steps that the computer performs, at which point the program ends. However, ActionScript programs are designed to keep running, waiting for user input or other things to happen. Events are the mechanism that determines which instructions the computer carries out and when.

In essence, *events* are things that happen that ActionScript is aware of and can respond to. Many events are related to user interaction, such as a user clicking a button or pressing a key on the keyboard. There are also other types of events. For example, if you use ActionScript to load an external image, there is an event that can let you know when the image has finished loading. When an ActionScript program is running, conceptually it just sits and waits for certain things to happen. When those things happen, the specific ActionScript code that you've specified for those events runs.

Basic event handling

The technique for specifying certain actions to perform in response to particular events is known as *event handling*. When you are writing ActionScript code to perform event handling, there are three important elements you'll want to identify:

- The event source: Which object is the one the event is going to happen to? For example, which button was clicked, or which Loader object is loading the image? The event source is also known as the *event target*. It has this name because it's the object where the computer targets the event (that is, where the event actually happens).
- The event: What is the thing that is going to happen, the thing that you want to respond to? The specific event is important to identify, because many objects trigger several events.
- The response: What steps do you want performed when the event happens?

Any time you write ActionScript code to handle events, it requires these three elements. The code follows this basic structure (elements in bold are placeholders you'd fill in for your specific case):

```
function eventResponse(eventObject:EventType):void
{
    // Actions performed in response to the event go here.
}

eventSource.addEventListener(EventType.EVENT_NAME, eventResponse);
```

This code does two things. First, it defines a function, which is the way to specify the actions you want performed in response to the event. Next, it calls the `addEventListener()` method of the source object. Calling `addEventListener()` essentially "subscribes" the function to the specified event. When the event happens, the function's actions are carried out. Consider each of these parts in more detail.

A *function* provides a way for you to group actions with a single name that is like a shortcut name to carry out the actions. A function is identical to a method except that it isn't necessarily associated with a specific class. (In fact, the term "method" could be defined as a function that is associated with a particular class.) When you're creating a function for event handling, you choose the name for the function (named `eventResponse` in this case). You also specify one parameter (named `eventObject` in this example). Specifying a function parameter is like declaring a variable, so you also have to indicate the data type of the parameter. (In this example, the parameter's data type is `EventType`.)

Each type of event that you want to listen to has an ActionScript class associated with it. The data type you specify for the function parameter is always the associated class of the specific event you want to respond to. For example, a `click` event (triggered when the user clicks an item with the mouse) is associated with the MouseEvent class. To write a listener function for a `click` event, you define the listener function with a parameter with the data type MouseEvent. Finally, between the opening and closing curly brackets ({ ... }), you write the instructions you want the computer to carry out when the event happens.

The event-handling function is written. Next you tell the event source object (the object that the event happens to, for example the button) that you want it to call your function when the event happens. You register your function with the event source object by calling the `addEventListener()` method of that object (all objects that have events also have an `addEventListener()` method). The `addEventListener()` method takes two parameters:

- First, the name of the specific event you want to respond to. Each event is affiliated with a specific class. Every event class has a special value, which is like a unique name, defined for each of its events. You use that value for the first parameter.
- Second, the name of your event response function. Note that a function name is written without parentheses when it's passed as a parameter.

The event-handling process

The following is a step-by-step description of the process that happens when you create an event listener. In this case, it's an example of creating a listener function that is called when an object named `myButton` is clicked.

The actual code written by the programmer is as follows:

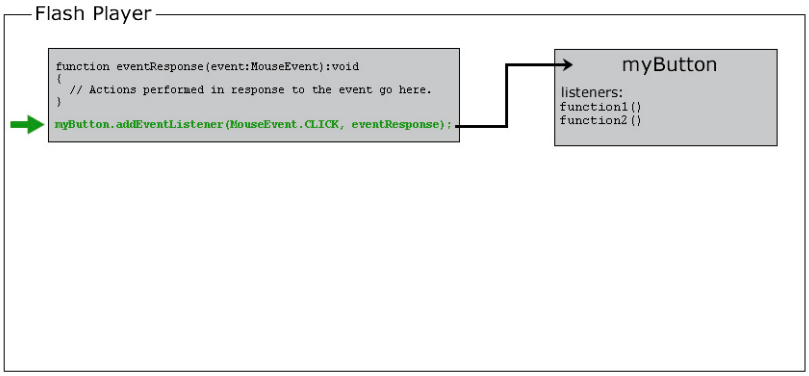
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

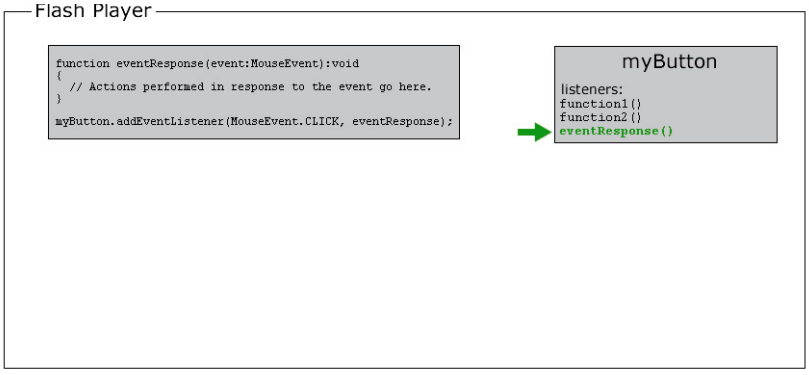
Here is how this code would actually work when it's running:

1 When the SWF file loads, the computer makes note of the fact that there's a function named `eventResponse()`.

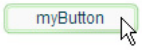
2 The computer then runs the code (specifically, the lines of code that aren't in a function). In this case that's only one line of code: calling the `addEventListener()` method on the event source object (named `myButton`) and passing the `eventResponse` function as a parameter.



Internally, `myButton` keeps a list of functions that are listening to each of its events. When its `addEventListener()` method is called, `myButton` stores the `eventResponse()` function in its list of event listeners.

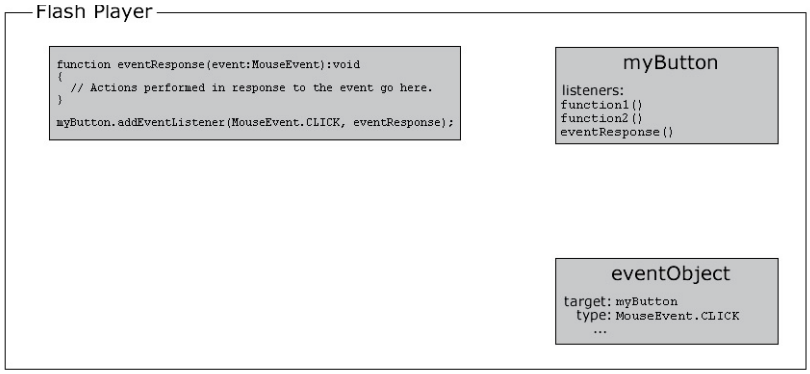


3 At some point, the user clicks the `myButton` object, triggering its `click` event (identified as `MouseEvent.CLICK` in the code).

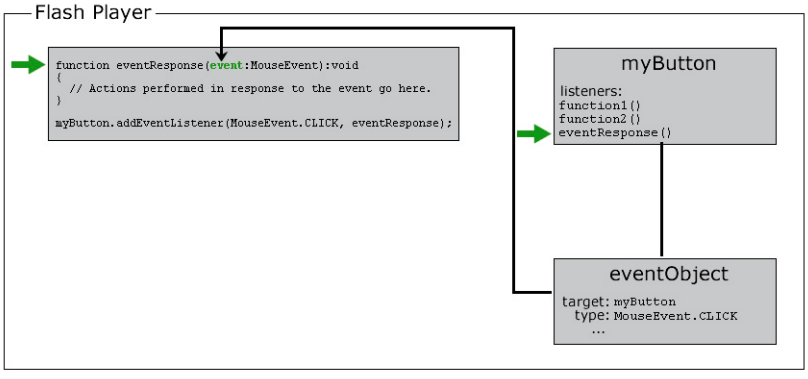


At that point, the following occurs:

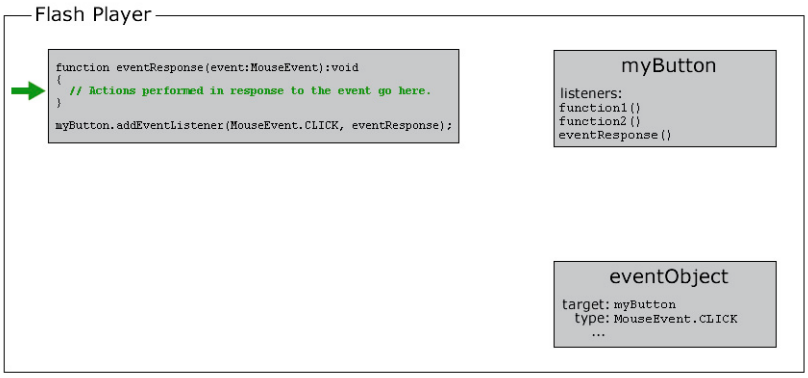
a An object is created that's an instance of the class associated with the event in question (MouseEvent in this example). For many events, this object is an instance of the Event class. For mouse events, it is a MouseEvent instance. For other events, it is an instance of the class that's associated with that event. This object that's created is known as the *event object*, and it contains specific information about the event that happened: what type of event it is, where it happened, and other event-specific information if applicable.



b The computer then looks at the list of event listeners stored by `myButton`. It goes through these functions one by one, calling each function and passing the event object to the function as a parameter. Since the `eventResponse()` function is one of `myButton`'s listeners, as part of this process the computer calls the `eventResponse()` function.



c When the `eventResponse()` function is called, the code in that function runs, so your specified actions are carried out.



Event-handling examples

Here are a few more concrete examples of event handling code. These examples are meant to give you an idea of some of the common event elements and possible variations available when you write event-handling code:

- Clicking a button to start the current movie clip playing. In the following example, `playButton` is the instance name of the button, and `this` is a special name meaning "the current object":

```
this.stop();

function playMovie(event:MouseEvent):void
{
    this.play();
}

playButton.addEventListener(MouseEvent.CLICK, playMovie);
```

- Detecting typing in a text field. In this example, `entryText` is an input text field, and `outputText` is a dynamic text field:

```
function updateOutput(event:TextEvent):void
{
    var pressedKey:String = event.text;
    outputText.text = "You typed: " + pressedKey;
}

entryText.addEventListener(TextEvent.TEXT_INPUT, updateOutput);
```

- Clicking a button to navigate to a URL. In this case, `linkButton` is the instance name of the button:

```
function gotoAdobeSite(event:MouseEvent):void
{
    var adobeURL:URLRequest = new URLRequest("http://www.adobe.com/");
    navigateToURL(adobeURL);
}

linkButton.addEventListener(MouseEvent.CLICK, gotoAdobeSite);
```

Creating object instances

Before you can use an object in ActionScript, the object has to exist in the first place. One part of creating an object is declaring a variable; however, declaring a variable only creates an empty place in the computer's memory. Always assign an actual value to the variable (create an object and store it in the variable) before you attempt to use or manipulate it. The process of creating an object is known as *instantiating* the object. In other words, you create an instance of a particular class.

One simple way to create an object instance doesn't involve ActionScript at all. In Flash Professional place a movie clip symbol, button symbol, or text field on the Stage and assign it an instance name. Flash Professional automatically declares a variable with that instance name, creates an object instance, and stores that object in the variable. Similarly, in Flex you create a component in MXML either by coding an MXML tag or by placing the component on the editor in Flash Builder Design mode. When you assign an ID to that component, that ID becomes the name of an ActionScript variable containing that component instance.

However, you don't always want to create an object visually, and for non-visual objects you can't. There are several additional ways you can create object instances using only ActionScript.

With several ActionScript data types, you can create an instance using a *literal expression*, which is a value written directly into the ActionScript code. Here are some examples:

- Literal numeric value (enter the number directly):

```
var someNumber:Number = 17.239;
var someNegativeInteger:int = -53;
var someUint:uint = 22;
```

- Literal String value (surround the text with double quotation marks):

```
var firstName:String = "George";
var soliloquy:String = "To be or not to be, that is the question...";
```

- Literal Boolean value (use the literal values `true` or `false`):

```
var niceWeather:Boolean = true;
var playingOutside:Boolean = false;
```

- Literal Array value (wrap a comma-separated list of values in square brackets):

```
var seasons:Array = ["spring", "summer", "autumn", "winter"];
```

- Literal XML value (enter the XML directly):

```
var employee:XML = <employee>
        <firstName>Harold</firstName>
        <lastName>Webster</lastName>
    </employee>;
```

ActionScript also defines literal expressions for the Array, RegExp, Object, and Function data types.

The most common way to create an instance for any data type is to use the `new` operator with the class name, as shown here:

```
var raceCar:MovieClip = new MovieClip();
var birthday:Date = new Date(2006, 7, 9);
```

Creating an object using the `new` operator is often described as "calling the class's constructor." A *constructor* is a special method that is called as part of the process of creating an instance of a class. Notice that when you create an instance in this way, you put parentheses after the class name. Sometimes you specify parameter values in the parentheses. These are two things that you also do when calling a method.

Even for those data types that let you create instances using a literal expression, you can also use the `new` operator to create an object instance. For example, these two lines of code do the same thing:

```
var someNumber:Number = 6.33;
var someNumber:Number = new Number(6.33);
```

It's important to be familiar with the `new` *ClassName*`()` way of creating objects. Many ActionScript data types don't have a visual representation. Consequently, they can't be created by placing an item on the Flash Professional Stage or the Design mode of Flash Builder's MXML editor. You can only create an instance of any of those data types in ActionScript using the `new` operator.

Adobe Flash Professional

In Flash Professional, the `new` operator can also be used to create an instance of a movie clip symbol that is defined in the Library but isn't placed on the Stage.

More Help topics

Working with arrays

Using regular expressions

Creating MovieClip objects with ActionScript

Common program elements

There are a few additional building blocks that you use to create an ActionScript program.

Operators

Operators are special symbols (or occasionally words) that are used to perform calculations. They are mostly used for math operations, and also used when comparing values to each other. Generally, an operator uses one or more values and "works out" to a single result. For example:

- The addition operator (`+`) adds two values together, resulting in a single number:

  ```
  var sum:Number = 23 + 32;
  ```

- The multiplication operator (`*`) multiplies one value by another, resulting in a single number:

  ```
  var energy:Number = mass * speedOfLight * speedOfLight;
  ```

- The equality operator (`==`) compares two values to see if they are equal, resulting in a single true-or-false (Boolean) value:

  ```
  if (dayOfWeek == "Wednesday")
  {
      takeOutTrash();
  }
  ```

  As shown here, the equality operator and the other "comparison" operators are most commonly used with the `if` statement to determine whether or not certain instructions are carried out.

Comments

As you're writing ActionScript, you'll often want to leave notes to yourself. For example, sometimes you want to explain how certain lines of code work or why you made a particular choice. *Code comments* are a tool you can use to write text that the computer ignores in your code. ActionScript includes two kinds of comments:

- Single-line comment: A single-line comment is designated by placing two slashes anywhere on a line. The computer ignores everything after the slashes up to the end of that line:

```
// This is a comment; it's ignored by the computer.
var age:Number = 10; // Set the age to 10 by default.
```

- Multiline comments: A multiline comment includes a starting comment marker (`/*`), then the comment content, and an ending comment marker (`*/`). The computer ignores everything between the starting and ending markers regardless of how many lines the comment spans:

```
/*
This is a long description explaining what a particular
function is used for or explaining a section of code.

In any case, the computer ignores these lines.
*/
```

Another common use of comments is to temporarily "turn off" one or more lines of code. For example, you can use comments if you're testing out a different way of doing something. You can also use them to try to figure out why certain ActionScript code isn't working the way you expect.

Flow control

Many times in a program, you want to repeat certain actions, perform only certain actions and not others, perform alternative actions depending on certain conditions, and so on. *Flow control* is the control over which actions are performed. There are several types of flow control elements available in ActionScript.

- Functions: Functions are like shortcuts. They provide a way to group a series of actions under a single name, and can be used to perform calculations. Functions are necessary for handling events, but are also used as a general tool for grouping a series of instructions.
- Loops: Loop structures let you designate a set of instructions that the computer performs a set number of times or until some condition changes. Often loops are used to manipulate several related items, using a variable whose value changes each time the computer works through the loop.
- Conditional statements: Conditional statements provide a way to designate certain instructions that are carried out only under certain circumstances. They are also used to provide alternative sets of instructions for different conditions. The most common type of conditional statement is the `if` statement. The `if` statement checks a value or expression in its parentheses. If the value is `true`, the lines of code in curly brackets are carried out. Otherwise, they are ignored. For example:

```
if (age < 20)
{
    // show special teenager-targeted content
}
```

The `if` statement's companion, the `else` statement, lets you designate alternative instructions that the computer performs if the condition is not `true`:

```
if (username == "admin")
{
    // do some administrator-only things, like showing extra options
}
else
{
    // do some non-administrator things
}
```

Example: Animation portfolio piece (Flash Professional)

This example is designed to give you a first opportunity to see how you can piece together bits of ActionScript into a complete application. The animation portfolio piece is an example of how you could take an existing linear animation and add some minor interactive elements. For example, you could incorporate an animation created for a client into an online portfolio. The interactive behavior that you'll add to the animation includes two buttons the viewer can click: one to start the animation, and one to navigate to a separate URL (such as the portfolio menu or the author's home page).

The process of creating this piece can be divided into these main sections:

1. Prepare the FLA file for adding ActionScript and interactive elements.
2. Create and add the buttons.
3. Write the ActionScript code.
4. Test the application.

Preparing to add interactivity

Before you can add interactive elements to your animation, it's helpful to set up the FLA file by creating some places to add your new content. This task includes creating actual space on the Stage where buttons can be placed. It also includes creating "space" in the FLA file for keeping different items separate.

To set up your FLA for adding interactive elements:

1. Create a FLA file with a simple animation such as a single motion tween or shape tween. If you already have a FLA file containing the animation that you're showcasing in the project, open that file and save it with a new name.
2. Decide where on the screen you'll want the two buttons to appear. One button is to start the animation and one is to link to the author portfolio or home page. If necessary, clear or add some space on the Stage for this new content. If the animation doesn't already have one, you can create a startup screen on the first frame. In that case you'll probably want to shift the animation over so it starts on Frame 2 or later.
3. Add a new layer, above the other layers in the Timeline, and name it **buttons**. This layer is where you'll add the buttons.
4. Add a new layer, above the buttons layer, and name it **actions**. This layer is where you'll add ActionScript code to your application.

Creating and adding buttons

Next, you'll actually create and position the buttons that form the center of the interactive application.

To create and add buttons to the FLA:

1. Using the drawing tools, create the visual appearance of your first button (the "play" button) on the buttons layer. For example, draw a horizontal oval with text on top of it.
2. Using the Selection tool, select all the graphic parts of the single button.
3. From the main menu, choose Modify > Convert To Symbol.
4. In the dialog box, choose Button as the symbol type, give the symbol a name, and click OK.
5. With the button selected, in the Property inspector give the button the instance name **playButton**.
6. Repeat steps 1 through 5 to create the button that takes the viewer to the author's home page. Name this button **homeButton**.

Writing the code

The ActionScript code for this application can be divided into three sets of functionality, although you enter it all in the same place. The three things the code does are:

- Stop the playhead as soon as the SWF file loads (when the playhead enters Frame 1).
- Listen for an event to start the SWF file playing when the user clicks the play button.
- Listen for an event to send the browser to the appropriate URL when the user clicks the author home page button.

To create code to stop the playhead when it enters Frame 1:

1. Select the keyframe on Frame 1 of the actions layer.
2. To open the Actions panel, from the main menu, choose Window > Actions.
3. In the Script pane, enter the following code:

```
stop();
```

To write code to start the animation when the play button is clicked:

1. At the end of the code entered in the previous steps, add two empty lines.
2. Enter the following code at the bottom of the script:

```
function startMovie(event:MouseEvent):void
{
    this.play();
}
```

   This code defines a function called `startMovie()`. When `startMovie()` is called, it causes the main timeline to start playing.

3. On the line following the code added in the previous step, enter this line of code:

```
playButton.addEventListener(MouseEvent.CLICK, startMovie);
```

   This line of code registers the `startMovie()` function as a listener for `playButton`'s `click` event. In other words, it makes it so that whenever the button named `playButton` is clicked, the `startMovie()` function is called.

To write code to send the browser to a URL when the home page button is clicked:

1. At the end of the code entered in the previous steps, add two empty lines.
2. Enter this code at the bottom of the script:

```
function gotoAuthorPage(event:MouseEvent):void
{
    var targetURL:URLRequest = new URLRequest("http://example.com/");
    navigateToURL(targetURL);
}
```

This code defines a function called `gotoAuthorPage()`. This function first creates a URLRequest instance representing the URL http://example.com/. It then passes that URL to the `navigateToURL()` function, causing the user's browser to open that URL.

3 On the line following the code added in the previous step, enter this line of code:

```
homeButton.addEventListener(MouseEvent.CLICK, gotoAuthorPage);
```

This line of code registers the `gotoAuthorPage()` function as a listener for `homeButton`'s `click` event. In other words, it makes it so that whenever the button named `homeButton` is clicked, the `gotoAuthorPage()` function is called.

Testing the application

The application is now completely functional. Let's test it to make sure that's the case.

To test the application:

1 From the main menu, choose Control > Test Movie. Flash Professional creates the SWF file and opens it in a Flash Player window.

2 Try both the buttons to make sure that they do what you expect them to.

3 If the buttons don't work, here are some things to check for:

- Do the buttons both have distinct instance names?
- Do the `addEventListener()` method calls use the same names as the buttons' instance names?
- Are the correct event names used in the `addEventListener()` method calls?
- Is the correct parameter specified for each of the functions? (Both methods need a single parameter with the data type MouseEvent.)

All of these mistakes and most other possible mistakes result in an error message. The error message can appear either when you choose the Test Movie command or when you click the button while testing the project. Look in the Compiler Errors panel for compiler errors (the ones that happen when you first choose Test Movie). Check the Output panel for run-time errors which happen while the content is playing, such as when you click a button.

Building applications with ActionScript

The process of writing ActionScript to build an application involves more than just knowing the syntax and the names of the classes you'll use. Most of the Flash Platform documentation covers those two topics (syntax and using ActionScript classes). However, to build an ActionScript application you'll also want to know information such as:

- What programs can be used for writing ActionScript?
- How do you organize ActionScript code?
- How do you include ActionScript code in an application?
- What steps do you follow in developing an ActionScript application?

Options for organizing your code

You can use ActionScript 3.0 code to power everything from simple graphics animations to complex client-server transaction processing systems. Depending on the type of application you're building, use one or more of these different ways of including ActionScript in your project.

Storing code in frames in a Flash Professional timeline

In Flash Professional, you can add ActionScript code to any frame in a timeline. This code executes while the movie is playing back, when the playhead enters that frame.

Placing ActionScript code in frames provides a simple way to add behaviors to applications built in Flash Professional. You can add code to any frame in the main timeline or to any frame in the timeline of any MovieClip symbol. However, this flexibility comes with a cost. When you build larger applications, it becomes easy to lose track of which frames contain which scripts. This complicated structure can make the application more difficult to maintain over time.

Many developers simplify the organization of their ActionScript code in the Flash Professional by placing code only in the first frame of a timeline or on a specific layer in the Flash document. Separating your code makes it easier to locate and maintain the code in your Flash FLA files. However, the same code can't be used in another Flash Professional project without copying and pasting the code into the new file.

To make it easier to use your ActionScript code in other Flash Professional projects in the future, store your code in external ActionScript files (text files with the .as extension).

Embedding code in Flex MXML files

In a Flex development environment such as Flash Builder, you can include ActionScript code inside an `<fx:Script>` tag in a Flex MXML file. However, this technique can add complexity to large projects and make it more difficult to use the same code in another Flex project. To make it easier to use your ActionScript code in other Flex projects in the future, store your code in external ActionScript files.

Note: *You can specify a source parameter for an* `<fx:Script>` *tag. Using a source parameter lets you "import" ActionScript code from an external file as if it was typed directly within the* `<fx:Script>` *tag. However, the source file that you use cannot define its own class, which limits its reusability.*

Storing code in ActionScript files

If your project involves significant ActionScript code, the best way to organize your code is in separate ActionScript source files (text files with the .as extension). An ActionScript file can be structured in one of two ways, depending on how you intend to use it in your application.

- Unstructured ActionScript code: Lines of ActionScript code, including statements or function definitions, written as though they were entered directly in a timeline script or MXML file.

  ActionScript written in this way can be accessed using the `include` statement in ActionScript, or the `<fx:Script>` tag in Flex MXML. The ActionScript `include` statement tells the compiler to include the contents of an external ActionScript file at a specific location and within a given scope in a script. The result is the same as if the code were entered there directly. In the MXML language, using an `<fx:Script>` tag with a source attribute identifies an external ActionScript that the compiler loads at that point in the application. For example, the following tag loads an external ActionScript file named Box.as:

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

- ActionScript class definition: A definition of an ActionScript class, including its method and property definitions.

When you define a class you can access the ActionScript code in the class by creating an instance of the class and using its properties, methods, and events. Using your own classes is identical to using any of the built-in ActionScript classes, and requires two parts:

- Use the `import` statement to specify the full name of the class, so the ActionScript compiler knows where to find it. For example, to use the MovieClip class in ActionScript, import the class using its full name, including package and class:

  ```
  import flash.display.MovieClip;
  ```

  Alternatively, you can import the package that contains the MovieClip class, which is equivalent to writing separate `import` statements for each class in the package:

  ```
  import flash.display.*;
  ```

  The top-level classes are the only exception to the rule that a class must be imported to use that class in your code. Those classes are not defined in a package.

- Write code that specifically uses the class name. For example, declare a variable with that class as its data type and create an instance of the class to store in the variable. By using a class in ActionScript code, you tell the compiler to load the definition of that class. For example, given an external class called Box, this statement creates an instance of the Box class:

  ```
  var smallBox:Box = new Box(10,20);
  ```

  The first time the compiler comes across the reference to the Box class it searches the available source code to locate the Box class definition.

Choosing the right tool

You can use one of several tools (or multiple tools together) for writing and editing your ActionScript code.

Flash Builder

Adobe Flash Builder is the premier tool for creating projects with the Flex framework or projects that primarily consist of ActionScript code. Flash Builder also includes a full-featured ActionScript editor as well as visual layout and MXML editing capabilities. It can be used to create Flex or ActionScript-only projects. Flex provides several benefits including a rich set of pre-built user interface controls, flexible dynamic layout controls, and built-in mechanisms for working with remote data and linking external data to user interface elements. However, because of the additional code required to provide these features, projects that use Flex can have a larger SWF file size than their non-Flex counterparts.

Use Flash Builder if you are creating full-featured data-driven rich Internet applications with Flex. Use it when you want to edit ActionScript code, edit MXML code, and lay out your application visually, all within a single tool.

Many Flash Professional users who build ActionScript-heavy projects use Flash Professional to create visual assets and Flash Builder as an editor for ActionScript code.

Flash Professional

In addition to its graphics and animation creation capabilities, Flash Professional includes tools for working with ActionScript code. The code can either be attached to elements in a FLA file or in external ActionScript-only files. Flash Professional is ideal for projects that involve significant animation or video. It is valuable when you want to create most of the graphic assets yourself. Another reason to use Flash Professional to develop your ActionScript project is to create visual assets and write code in the same application. Flash Professional also includes pre-built user interface components. You can use those components to achieve smaller SWF file size and use visual tools to skin them for your project.

Flash Professional includes two tools for writing ActionScript code:

- Actions panel: Available when working in a FLA file, this panel allows you to write ActionScript code attached to frames on a timeline.
- Script window: The Script window is a dedicated text editor for working with ActionScript (.as) code files.

Third-party ActionScript editor

Because ActionScript (.as) files are stored as simple text files, any program that can edit plain text files can be used to write ActionScript files. In addition to Adobe's ActionScript products, several third-party text editing programs with ActionScript-specific capabilities have been created. You can write an MXML file or ActionScript classes using any text editor program. You can then create an application from those files using the Flex SDK. The project can use Flex or be an ActionScript-only application. Alternatively, some developers use Flash Builder or a third-party ActionScript editor for writing ActionScript classes, in combination with Flash Professional for creating graphical content.

Reasons to choose a third-party ActionScript editor include:

- You prefer to write ActionScript code in a separate program and design visual elements in Flash Professional.
- You use an application for non-ActionScript programming (such as creating HTML pages or building applications in another programming language). You want to use the same application for your ActionScript coding as well.
- You want to create ActionScript-only or Flex projects using the Flex SDK without Flash Professional or Flash Builder.

Some of the notable code editors providing ActionScript-specific support include:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (with ActionScript and Flex bundles)

The ActionScript development process

Whether your ActionScript project is large or small, using a process to design and develop your application makes work more efficient and effective. The following steps describe a basic development process for building an application that uses ActionScript 3.0:

1. Design your application.

   Describe your application in some way before you start building it.

2. Compose your ActionScript 3.0 code.

   You can create ActionScript code using Flash Professional, Flash Builder, Dreamweaver, or a text editor.

3. Create a Flash or Flex project to run your code.

   In Flash Professional, create a FLA file, set up the publish settings, add user interface components to the application, and reference the ActionScript code. In Flex, define the application, add user interface components using MXML, and reference the ActionScript code.

4. Publish and test your ActionScript application.

Testing your application involves running your application from within your development environment and making sure that it does everything you intended.

You don't necessarily have to follow these steps in order, or completely finish one step before working on another. For example, you can design one screen of your application (step 1) and then create the graphics, buttons, and so on (step 3) before writing ActionScript code (step 2) and testing (step 4). Or you can design part of it and then add one button or interface element at a time, writing ActionScript for each one and testing it as it's built. It's helpful to remember these four stages of the development process. However, in real-world development it's more effective to move back and forth among the stages as appropriate.

Creating your own classes

The process of creating classes for use in your projects can seem daunting. However, the more difficult part of creating a class is the task of designing the class's methods, properties, and events.

Strategies for designing a class

The topic of object-oriented design is a complex one; entire careers have been devoted to the academic study and professional practice of this discipline. Nevertheless, here are a few suggested approaches that can help you get started.

1. Think about the role that the instances of this class play in the application. Generally, objects serve one of these three roles:
   - Value object: These objects serve primarily as containers of data. They probably have several properties and fewer methods (or sometimes no methods). They are generally code representations of clearly defined items. For example, a music player application could include a Song class representing a single real-world song and a Playlist class representing a conceptual group of songs.
   - Display object: These are objects that actually appear on the screen. Examples include user-interface elements like a drop-down list or status readout, graphical elements like creatures in a video game, and so on.
   - Application structure: These objects play a broad range of supporting roles in the logic or processing performed by applications. For example, you can make an object to perform certain calculations in a biology simulation. You can make one that's responsible for synchronizing values between a dial control and a volume readout in a music player application. Another one can manage the rules in a video game. Or you can make a class to load a saved picture in a drawing application.
2. Decide the specific functionality that the class needs. The different types of functionality often become the methods of the class.
3. If the class is intended to serve as a value object, decide the data that the instances include. These items are good candidates for properties.
4. Since your class is being designed specifically for your project, what's most important is that you provide the functionality that your application needs. Try to answer these questions for yourself:
   - What pieces of information is your application storing, tracking, and manipulating? Answering this question helps you identify any value objects and properties you need.
   - What sets of actions does the application perform? For example, what happens when the application first loads, when a particular button is clicked, when a movie stops playing, and so on? These are good candidates for methods. They can also be properties if the "actions" involve changing individual values.
   - For any given action, what information is necessary to perform that action? Those pieces of information become the parameters of the method.

- As the application proceeds to do its work, what things change in your class that other parts of your application need to know about? These are good candidates for events.

5 Is there is an existing object that is similar to the object you need except that it's lacking some additional functionality you want to add? Consider creating a subclass. (A *subclass* is a class which builds on the functionality of an existing class, rather than defining all of its own functionality.) For example, to create a class that is a visual object on the screen, use the behavior of an existing display object as a basis for your class. In that case, the display object (such as MovieClip or Sprite) would be the *base class*, and your class would extend that class.

Writing the code for a class

Once you have a design for your class, or at least some idea of what information it stores and what actions it carries out, the actual syntax of writing a class is fairly straightforward.

Here are the minimum steps to create your own ActionScript class:

1 Open a new text document in your ActionScript text editor program.

2 Enter a `class` statement to define the name of the class. To add a `class` statement, enter the words `public class` and then the class's name. Add opening and closing curly brackets to contain the contents of the class (the method and property definitions). For example:

```
public class MyClass
{
}
```

The word `public` indicates that the class can be accessed from any other code. For other alternatives, see Access control namespace attributes.

3 Type a `package` statement to indicate the name of the package that contains your class. The syntax is the word `package`, followed by the full package name, followed by opening and closing curly brackets around the `class` statement block) For example, change the code in the previous step to the following:

```
package mypackage
{
    public class MyClass
    {
    }
}
```

4 Define each property in the class using the `var` statement within the class body. The syntax is the same as you use to declare any variable (with the addition of the `public` modifier). For example, adding these lines between the opening and closing curly brackets of the class definition creates properties named `textProperty`, `numericProperty`, and `dateProperty`:

```
public var textProperty:String = "some default value";
public var numericProperty:Number = 17;
public var dateProperty:Date;
```

5 Define each method in the class using the same syntax that's used to define a function. For example:

- To create a `myMethod()` method, enter:

```
public function myMethod(param1:String, param2:Number):void
{
    // do something with parameters
}
```

- To create a constructor (the special method that is called as part of the process of creating an instance of a class), create a method whose name matches exactly the name of the class:

```
public function MyClass()
{
    // do stuff to set initial values for properties
    // and otherwise set up the object
    textVariable = "Hello there!";
    dateVariable = new Date(2001, 5, 11);
}
```

If you don't include a constructor method in your class, the compiler automatically creates an empty constructor in your class. (In other words, a constructor with no parameters and no statements.)

There are a few more class elements that you can define. These elements are more complex.

- *Accessors* are a special cross between a method and a property. When you write the code to define the class, you write the accessor like a method. You can perform multiple actions rather than just reading or assigning a value, which is all you can do when you define a property. However, when you create an instance of your class, you treat the accessor like a property and use the name to read or assign the value.
- Events in ActionScript aren't defined using a specific syntax. Instead, you define events in your class using the functionality of the EventDispatcher class.

More Help topics

Handling events

Example: Creating a basic application

ActionScript 3.0 can be used within a number of application development environments, including the Flash Professional and Flash Builder tools or any text editor.

This example walks through the steps in creating and enhancing a simple ActionScript 3.0 application using Flash Professional or Flash Builder. The application you'll build presents a simple pattern for using external ActionScript 3.0 class files in Flash Professional and Flex.

Designing your ActionScript application

This example ActionScript application is a standard "Hello World" application, so its design is simple:

- The application is called HelloWorld.
- It displays a single text field containing the words "Hello World!"
- The application uses a single object-oriented class named Greeter. This design allows the class to be used from within a Flash Professional or Flex project.
- In this example, you first create a basic version of the application. Next you add functionality to have the user enter a user name and have the application check the name against a list of known users.

With that concise definition in place, you can start building the application itself.

Creating the HelloWorld project and the Greeter class

The design statement for the Hello World application says that its code is easy to reuse. To achieve that goal, the application uses a single object-oriented class named Greeter. You use that class from within an application that you create in Flash Builder or Flash Professional.

To create the HelloWorld project and Greeter class in Flex:

1 In Flash Builder, select File > New> Flex Project,

2 Type HelloWorld as the Project Name. Make sure that the Application type is set to "Web (runs in Adobe Flash Player)," and then click Finish.

Flash Builder creates your project and displays it in the Package Explorer. By default the project already contains a file named HelloWorld.mxml, and that file is open in the editor.

3 Now to create a custom ActionScript class file in Flash Builder, select File > New > ActionScript Class.

4 In the New ActionScript Class dialog box, in the Name field, type **Greeter** as the class name, and then click Finish.

A new ActionScript editing window is displayed.

Continue with adding code to the Greeter class.

To create the Greeter class in Flash Professional:

1 In Flash Professional, select File > New.

2 In the New Document dialog box, select ActionScript file, and click OK.

A new ActionScript editing window is displayed.

3 Select File > Save. Select a folder to contain your application, name the ActionScript file **Greeter.as**, and then click OK.

Continue with adding code to the Greeter class.

Adding code to the Greeter class

The Greeter class defines an object, `Greeter`, that you use in your HelloWorld application.

To add code to the Greeter class:

1 Type the following code into the new file (some of the code may have been added for you):

```
package
{
    public class Greeter
    {
        public function sayHello():String
        {
            var greeting:String;
            greeting = "Hello World!";
            return greeting;
        }
    }
}
```

The Greeter class includes a single `sayHello()` method, which returns a string that says "Hello World!".

2 Select File > Save to save this ActionScript file.

The Greeter class is now ready to be used in an application.

Creating an application that uses your ActionScript code

The Greeter class that you have built defines a self-contained set of software functions, but it does not represent a complete application. To use the class, you create a Flash Professional document or Flex project.

The code needs an instance of the Greeter class. Here's how to use the Greeter class to your application.

To create an ActionScript application using Flash Professional:

1 Select File > New.

2 In the New Document dialog box, select Flash File (ActionScript 3.0), and click OK.

A new document window is displayed.

3 Select File > Save. Select the same folder that contains the Greeter.as class file, name the Flash document **HelloWorld.fla**, and click OK.

4 In the Flash Professional tools palette, select the Text tool. Drag across the Stage to define a new text field approximately 300 pixels wide and 100 pixels high.

5 In the Properties panel, with the text field still selected on the Stage, set the text type to "Dynamic Text." Type **mainText** as the instance name of the text field.

6 Click the first frame of the main timeline. Open the Actions panel by choosing Window > Actions.

7 In the Actions panel, type the following script:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello();
```

8 Save the file.

Continue with Publishing and testing your ActionScript application.

To create an ActionScript application using Flash Builder:

1 Open the HelloWorld.mxml file, and add code to match the following listing:

```
<?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/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400"/>

</s:Application>
```

This Flex project includes four MXML tags:

- An `<s:Application>` tag, which defines the Application container

- An `<s:layout>` tag, which defines the layout style (vertical layout) for the Application tag
- An `<fx:Script>` tag that includes some ActionScript code
- An `<s:TextArea>` tag, which defines a field to display text messages to the user

The code in the `<fx:Script>` tag defines an `initApp()` method that is called when the application loads. The `initApp()` method sets the text value of the `mainTxt` TextArea to the "Hello World!" string returned by the `sayHello()` method of the custom class Greeter, which you just wrote.

2 Select File > Save to save the application.

Continue with Publishing and testing your ActionScript application.

Publishing and testing your ActionScript application

Software development is an iterative process. You write some code, try to compile it, and edit the code until it compiles cleanly. You run the compiled application and test it to see if it fulfills the intended design. If it doesn't, you edit the code again until it does. The Flash Professional and Flash Builder development environments offer a number of ways to publish, test, and debug your applications.

Here are the basic steps for testing the HelloWorld application in each environment.

To publish and test an ActionScript application using Flash Professional:

1 Publish your application and watch for compilation errors. In Flash Professional, select Control > Test Movie to compile your ActionScript code and run the HelloWorld application.

2 If any errors or warnings are displayed in the Output window when you test your application, fix these errors in the HelloWorld.fla or HelloWorld.as files. Then try testing the application again.

3 If there are no compilation errors, you see a Flash Player window showing the Hello World application.

You have created a simple but complete object-oriented application that uses ActionScript 3.0. Continue with Enhancing the HelloWorld application.

To publish and test an ActionScript application using Flash Builder:

1 Select Run > Run HelloWorld.

2 The HelloWorld application starts.

- If any errors or warnings are displayed in the Output window when you test your application, fix the errors in the HelloWorld.mxml or Greeter.as files. Then try testing the application again.
- If there are no compilation errors, a browser window opens showing the Hello World application. The text "Hello World!" appears.

You have created a simple but complete object-oriented application that uses ActionScript 3.0. Continue with Enhancing the HelloWorld application.

Enhancing the HelloWorld application

To make the application a little more interesting, you'll now make it ask for and validate a user name against a predefined list of names.

First you update the Greeter class to add new functionality. Then you update the application to use the new functionality.

To update the Greeter.as file:

1 Open the Greeter.as file.

2 Change the contents of the file to the following (new and changed lines are shown in boldface):

```
package
{
    public class Greeter
    {
        /**
         * Defines the names that receive a proper greeting.
         */
        public static var validNames:Array = ["Sammy", "Frank", "Dean"];

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                            + "the Enter key.";
            }
            else if (validName(userName))
            {
                greeting = "Hello, " + userName + ".";
            }
            else
            {
                greeting = "Sorry " + userName + ", you are not on the list.";
            }
            return greeting;
        }

        /**
         * Checks whether a name is in the validNames list.
         */
        public static function validName(inputName:String = ""):Boolean
        {
            if (validNames.indexOf(inputName) > -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

The Greeter class now has a number of new features:

- The `validNames` array lists valid user names. The array is initialized to a list of three names when the Greeter class is loaded.

- The `sayHello()` method now accepts a user name and changes the greeting based on some conditions. If the `userName` is an empty string (`""`), the `greeting` property is set to prompt the user for a name. If the user name is valid, the greeting becomes `"Hello, `*userName*`."` Finally, if either of those two conditions are not met, the `greeting` variable is set to `"Sorry `*userName*`, you are not on the list."`
- The `validName()` method returns `true` if the `inputName` is found in the `validNames` array, and `false` if it is not found. The statement `validNames.indexOf(inputName)` checks each of the strings in the `validNames` array against the `inputName` string. The `Array.indexOf()` method returns the index position of the first instance of an object in an array. It returns the value -1 if the object is not found in the array.

Next you edit the application file that references this ActionScript class.

To modify the application using Flash Professional:

1. Open the HelloWorld.fla file.
2. Modify the script in Frame 1 so that an empty string (`""`) is passed to the Greeter class's `sayHello()` method:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```

3. Select the Text tool in the tools palette. Create two new text fields on the Stage. Place them side by side directly under the existing `mainText` text field.
4. In the first new text field, which is the label, type the text **User Name:**.
5. Select the other new text field, and in the Property inspector, select Input Text as the type of text field. Select Single line as the Line type. Type **textIn** as the instance name.
6. Click the first frame of the main timeline.
7. In the Actions panel, add the following lines to the end of the existing script:

```
mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

The new code adds the following functionality:

- The first two lines simply define borders for two text fields.
- An input text field, such as the `textIn` field, has a set of events that it can dispatch. The `addEventListener()` method lets you define a function that runs when a type of event occurs. In this case, that event is the pressing of a key on the keyboard.
- The `keyPressed()` custom function checks if the key that was pressed is the Enter key. If so it calls the `sayHello()` method of the `myGreeter` object, passing the text from the `textIn` text field as a parameter. That method returns a string greeting based on the value passed in. The returned string is then assigned to the `text` property of the `mainText` text field.

The complete script for Frame 1 is the following:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");

mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

8 Save the file.

9 Select Control > Test Movie to run the application.

When you run the application, it prompts you to enter a user name. If it is valid (Sammy, Frank, or Dean), the application displays the "hello" confirmation message.

To modify the application using Flash Builder:

1 Open the HelloWorld.mxml file.

2 Next modify the `<mx:TextArea>` tag to indicate to the user that it is for display only. Change the background color to a light gray and set the `editable` attribute to `false`:

```
<s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false" />
```

3 Now add the following lines right after the `<s:TextArea>` closing tag. These lines create a TextInput component that lets the user enter a user name value:

```
<s:HGroup width="400">
    <mx:Label text="User Name:"/>
    <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
</s:HGroup>
```

The `enter` attribute defines what happens when the user presses the Enter key in the `userNameTxt` field. In this example, the code passes the text in the field to the `Greeter.sayHello()` method. The greeting in the `mainTxt` field changes accordingly.

The HelloWorld.mxml file looks like the following:

```
<?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/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false"/>

    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>

</s:Application>
```

4 Save the edited HelloWorld.mxml file. Select Run > Run HelloWorld to run the application.

When you run the application, the application prompts you to enter a user name. If it is valid (Sammy, Frank, or Dean), the application displays the "`Hello, `*userName*" confirmation message.

Chapter 3: ActionScript language and syntax

ActionScript 3.0 includes both the core ActionScript language and the Adobe Flash Platform Application Programming Interface (API). The core language is the part of ActionScript that defines the syntax of the language as well as the top-level data types. ActionScript 3.0 provides programmatic access to the Adobe Flash Platform runtimes: Adobe Flash Player and Adobe AIR.

Language overview

Objects lie at the heart of the ActionScript 3.0 language—they are its fundamental building blocks. Every variable you declare, every function you write, and every class instance you create is an object. You can think of an ActionScript 3.0 program as a group of objects that carry out tasks, respond to events, and communicate with one another.

Programmers familiar with object-oriented programming (OOP) in Java or C++ may think of objects as modules that contain two kinds of members: data stored in member variables or properties, and behavior accessible through methods. ActionScript 3.0 defines objects in a similar but slightly different way. In ActionScript 3.0, objects are simply collections of properties. These properties are containers that can hold not only data, but also functions or other objects. If a function is attached to an object in this way, it is called a method.

While the ActionScript 3.0 definition may seem a little odd to programmers with a Java or C++ background, in practice, defining object types with ActionScript 3.0 classes is very similar to the way classes are defined in Java or C++. The distinction between the two definitions of object is important when discussing the ActionScript object model and other advanced topics, but in most other situations the term *properties* means class member variables as opposed to methods. ActionScript 3.0 Reference for the Adobe Flash Platform, for example, uses the term *properties* to mean variables or getter-setter properties. It uses the term *methods* to mean functions that are part of a class.

One subtle difference between classes in ActionScript and classes in Java or C++ is that in ActionScript, classes are not just abstract entities. ActionScript classes are represented by *class objects* that store the class's properties and methods. This allows for techniques that may seem alien to Java and C++ programmers, such as including statements or executable code at the top level of a class or package.

Another difference between ActionScript classes and Java or C++ classes is that every ActionScript class has something called a *prototypeobject*. In previous versions of ActionScript, prototype objects, linked together into *prototype chains,* served collectively as the foundation of the entire class inheritance hierarchy. In ActionScript 3.0, however, prototype objects play only a small role in the inheritance system. The prototype object can still be useful, however, as an alternative to static properties and methods if you want to share a property and its value among all the instances of a class.

In the past, advanced ActionScript programmers could directly manipulate the prototype chain with special built-in language elements. Now that the language provides a more mature implementation of a class-based programming interface, many of these special language elements, such as `__proto__` and `__resolve`, are no longer part of the language. Moreover, optimizations of the internal inheritance mechanism that provide significant performance improvements preclude direct access to the inheritance mechanism.

Objects and classes

In ActionScript 3.0, every object is defined by a class. A class can be thought of as a template or a blueprint for a type of object. Class definitions can include variables and constants, which hold data values, and methods, which are functions that encapsulate behavior bound to the class. The values stored in properties can be *primitive values* or other objects. Primitive values are numbers, strings, or Boolean values.

ActionScript contains a number of built-in classes that are part of the core language. Some of these built-in classes, such as Number, Boolean and String, represent the primitive values available in ActionScript. Others, such as the Array, Math, and XML classes, define more complex objects.

All classes, whether built-in or user-defined, derive from the Object class. For programmers with previous ActionScript experience, it is important to note that the Object data type is no longer the default data type, even though all other classes still derive from it. In ActionScript 2.0, the following two lines of code were equivalent because the lack of a type annotation meant that a variable would be of type Object:

```
var someObj:Object;
var someObj;
```

ActionScript 3.0, however, introduces the concept of untyped variables, which can be designated in the following two ways:

```
var someObj:*;
var someObj;
```

An untyped variable is not the same as a variable of type Object. The key difference is that untyped variables can hold the special value `undefined`, while a variable of type Object cannot hold that value.

You can define your own classes using the `class` keyword. You can declare class properties in three ways: constants can be defined with the `const` keyword, variables are defined with the `var` keyword, and getter and setter properties are defined by using the `get` and `set` attributes in a method declaration. You can declare methods with the `function` keyword.

You create an instance of a class by using the `new` operator. The following example creates an instance of the Date class called `myBirthday`.

```
var myBirthday:Date = new Date();
```

Packages and namespaces

Packages and namespaces are related concepts. Packages allow you to bundle class definitions together in a way that facilitates code sharing and minimizes naming conflicts. Namespaces allow you to control the visibility of identifiers, such as property and method names, and can be applied to code whether it resides inside or outside a package. Packages let you organize your class files, and namespaces let you manage the visibility of individual properties and methods.

Packages

Packages in ActionScript 3.0 are implemented with namespaces, but are not synonymous with them. When you declare a package, you are implicitly creating a special type of namespace that is guaranteed to be known at compile time. Namespaces, when created explicitly, are not necessarily known at compile time.

The following example uses the `package` directive to create a simple package containing one class:

```
package samples
{
    public class SampleCode
    {
        public var sampleGreeting:String;
        public function sampleFunction()
        {
            trace(sampleGreeting + " from sampleFunction()");
        }
    }
}
```

The name of the class in this example is SampleCode. Because the class is inside the samples package, the compiler automatically qualifies the class name at compile time into its fully qualified name: samples.SampleCode. The compiler also qualifies the names of any properties or methods, so that `sampleGreeting` and `sampleFunction()` become `samples.SampleCode.sampleGreeting` and `samples.SampleCode.sampleFunction()`, respectively.

Many developers, especially those with Java programming backgrounds, may choose to place only classes at the top level of a package. ActionScript 3.0, however, supports not only classes at the top level of a package, but also variables, functions, and even statements. One advanced use of this feature is to define a namespace at the top level of a package so that it is available to all classes in that package. Note, however, that only two access specifiers, `public` and `internal`, are allowed at the top level of a package. Unlike Java, which allows you to declare nested classes as private, ActionScript 3.0 doesn't support nested or private classes.

In many other ways, however, ActionScript 3.0 packages are similar to packages in the Java programming language. As you can see in the previous example, fully qualified package references are expressed using the dot operator (.), just as they are in Java. You can use packages to organize your code into an intuitive hierarchical structure for use by other programmers. This facilitates code sharing by allowing you to create your own package to share with others, and to use packages created by others in your code.

The use of packages also helps to ensure that the identifier names that you use are unique and do not conflict with other identifier names. In fact, some would argue that this is the primary benefit of packages. For example, two programmers who want to share their code with each other may have each created a class called SampleCode. Without packages, this would create a name conflict, and the only resolution would be to rename one of the classes. With packages, however, the name conflict is easily avoided by placing one, or preferably both, of the classes in packages with unique names.

You can also include embedded dots in your package name to create nested packages. This allows you to create a hierarchical organization of packages. A good example of this is the flash.display package provided by ActionScript 3.0. The flash.display package is nested inside the flash package.

Most of ActionScript 3.0 is organized under the flash package. For example, the flash.display package contains the display list API, and the flash.events package contains the new event model.

Creating packages

ActionScript 3.0 provides significant flexibility in the way you organize your packages, classes, and source files. Previous versions of ActionScript allowed only one class per source file and required the name of the source file to match the name of the class. ActionScript 3.0 allows you to include multiple classes in one source file, but only one class in each file can be made available to code that is external to that file. In other words, only one class in each file can be declared inside a package declaration. You must declare any additional classes outside your package definition, which makes those classes invisible to code outside that source file. The name of the class declared inside the package definition must match the name of the source file.

ActionScript 3.0 also provides more flexibility in the way you declare packages. In previous versions of ActionScript, packages merely represented directories in which you placed source files, and you didn't declare packages with the `package` statement, but rather included the package name as part of the fully qualified class name in your class declaration. Although packages still represent directories in ActionScript 3.0, packages can contain more than just classes. In ActionScript 3.0, you use the `package` statement to declare a package, which means that you can also declare variables, functions, and namespaces at the top level of a package. You can even include executable statements at the top level of a package. If you do declare variables, functions, or namespaces at the top level of a package, the only attributes available at that level are `public` and `internal`, and only one package-level declaration per file can use the `public` attribute, whether that declaration is a class, variable, function, or namespace.

Packages are useful for organizing your code and for preventing name conflicts. You should not confuse the concept of packages with the unrelated concept of class inheritance. Two classes that reside in the same package have a namespace in common, but are not necessarily related to each other in any other way. Likewise, a nested package may have no semantic relationship to its parent package.

Importing packages

If you want to use a class that is inside a package, you must import either the package or the specific class. This differs from ActionScript 2.0, where importing classes was optional.

For example, consider the SampleCode class example presented previously. If the class resides in a package named samples, you must use one of the following import statements before using the SampleCode class:

```
import samples.*;
```

or

```
import samples.SampleCode;
```

In general, `import` statements should be as specific as possible. If you plan to use only the SampleCode class from the samples package, you should import only the SampleCode class rather than the entire package to which it belongs. Importing entire packages may lead to unexpected name conflicts.

You must also place the source code that defines the package or class within your *classpath*. The classpath is a user-defined list of local directory paths that determines where the compiler searches for imported packages and classes. The classpath is sometimes called the *build path* or *source path*.

After you have properly imported the class or package, you can use either the fully qualified name of the class (samples.SampleCode) or merely the class name by itself (SampleCode).

Fully qualified names are useful when identically named classes, methods, or properties result in ambiguous code, but can be difficult to manage if used for all identifiers. For example, the use of the fully qualified name results in verbose code when you instantiate a SampleCode class instance:

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

As the levels of nested packages increase, the readability of your code decreases. In situations where you are confident that ambiguous identifiers are not a problem, you can make your code easier to read by using simple identifiers. For example, instantiating a new instance of the SampleCode class is much less verbose if you use only the class identifier:

```
var mySample:SampleCode = new SampleCode();
```

If you attempt to use identifier names without first importing the appropriate package or class, the compiler can't find the class definitions. On the other hand, if you do import a package or class, any attempt to define a name that conflicts with an imported name generates an error.

When a package is created, the default access specifier for all members of that package is `internal`, which means that, by default, package members are only visible to other members of that package. If you want a class to be available to code outside the package, you must declare that class to be `public`. For example, the following package contains two classes, SampleCode and CodeFormatter:

```
// SampleCode.as file
package samples
{
    public class SampleCode {}
}

// CodeFormatter.as file
package samples
{
    class CodeFormatter {}
}
```

The SampleCode class is visible outside the package because it is declared as a `public` class. The CodeFormatter class, however, is visible only within the samples package itself. If you attempt to access the CodeFormatter class outside the samples package, it generates an error, as the following example shows:

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // okay, public class
var myFormatter:CodeFormatter = new CodeFormatter(); // error
```

If you want both classes to be available outside the package, you must declare both classes to be `public`. You cannot apply the `public` attribute to the package declaration.

Fully qualified names are useful for resolving name conflicts that may occur when using packages. Such a scenario may arise if you import two packages that define classes with the same identifier. For example, consider the following package, which also has a class named SampleCode:

```
package langref.samples
{
    public class SampleCode {}
}
```

If you import both classes, as follows, you have a name conflict when using the SampleCode class:

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // name conflict
```

The compiler has no way of knowing which SampleCode class to use. To resolve this conflict, you must use the fully qualified name of each class, as follows:

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

Note: *Programmers with a C++ background often confuse the `import` statement with `#include`. The `#include` directive is necessary in C++ because C++ compilers process one file at a time, and don't look in other files for class definitions unless a header file is explicitly included. ActionScript 3.0 has an `include` directive, but it is not designed to import classes and packages. To import classes or packages in ActionScript 3.0, you must use the `import` statement and place the source file that contains the package in the class path.*

Namespaces

Namespaces give you control over the visibility of the properties and methods that you create. Think of the `public`, `private`, `protected`, and `internal` access control specifiers as built-in namespaces. If these predefined access control specifiers do not suit your needs, you can create your own namespaces.

If you are familiar with XML namespaces, much of this discussion will not be new to you, although the syntax and details of the ActionScript implementation are slightly different from those of XML. If you have never worked with namespaces before, the concept itself is straightforward, but the implementation has specific terminology that you will need to learn.

To understand how namespaces work, it helps to know that the name of a property or method always contains two parts: an identifier and a namespace. The identifier is what you generally think of as a name. For example, the identifiers in the following class definition are `sampleGreeting` and `sampleFunction()`:

```
class SampleCode
{
    var sampleGreeting:String;
    function sampleFunction () {
        trace(sampleGreeting + " from sampleFunction()");
    }
}
```

Whenever definitions are not preceded by a namespace attribute, their names are qualified by the default `internal` namespace, which means they are visible only to callers in the same package. If the compiler is set to strict mode, the compiler issues a warning that the `internal` namespace applies to any identifier without a namespace attribute. To ensure that an identifier is available everywhere, you must specifically precede the identifier name with the `public` attribute. In the previous example code, both `sampleGreeting` and `sampleFunction()` have a namespace value of `internal`.

There are three basic steps to follow when using namespaces. First, you must define the namespace using the `namespace` keyword. For example, the following code defines the `version1` namespace:

```
namespace version1;
```

Second, you apply your namespace by using it instead of an access control specifier in a property or method declaration. The following example places a function named `myFunction()` into the `version1` namespace:

```
version1 function myFunction() {}
```

Third, once you've applied the namespace, you can reference it with the `use` directive or by qualifying the name of an identifier with a namespace. The following example references the `myFunction()` function through the `use` directive:

```
use namespace version1;
myFunction();
```

You can also use a qualified name to reference the `myFunction()` function, as the following example shows:

```
version1::myFunction();
```

Defining namespaces

Namespaces contain one value, the Uniform Resource Identifier (URI), which is sometimes called the *namespace name*. A URI allows you to ensure that your namespace definition is unique.

You create a namespace by declaring a namespace definition in one of two ways. You can either define a namespace with an explicit URI, as you would define an XML namespace, or you can omit the URI. The following example shows how a namespace can be defined using a URI:

```
namespace flash_proxy = "http://www.adobe.com/flash/proxy";
```

The URI serves as a unique identification string for that namespace. If you omit the URI, as in the following example, the compiler creates a unique internal identification string in place of the URI. You do not have access to this internal identification string.

```
namespace flash_proxy;
```

Once you define a namespace, with or without a URI, that namespace cannot be redefined in the same scope. An attempt to define a namespace that has been defined earlier in the same scope results in a compiler error.

If a namespace is defined within a package or a class, the namespace may not be visible to code outside that package or class unless the appropriate access control specifier is used. For example, the following code shows the `flash_proxy` namespace defined within the flash.utils package. In the following example, the lack of an access control specifier means that the `flash_proxy` namespace would be visible only to code within the flash.utils package and would not be visible to any code outside the package:

```
package flash.utils
{
    namespace flash_proxy;
}
```

The following code uses the `public` attribute to make the `flash_proxy` namespace visible to code outside the package:

```
package flash.utils
{
    public namespace flash_proxy;
}
```

Applying namespaces

Applying a namespace means placing a definition into a namespace. Definitions that can be placed into namespaces include functions, variables, and constants (you cannot place a class into a custom namespace).

Consider, for example, a function declared using the `public` access control namespace. Using the `public` attribute in a function definition places the function into the public namespace, which makes the function available to all code. Once you have defined a namespace, you can use the namespace that you defined the same way you would use the `public` attribute, and the definition is available to code that can reference your custom namespace. For example, if you define a namespace `example1`, you can add a method called `myFunction()` using `example1` as an attribute, as the following example shows:

```
namespace example1;
class someClass
{
    example1 myFunction() {}
}
```

Declaring the `myFunction()` method using the namespace `example1` as an attribute means that the method belongs to the `example1` namespace.

You should bear in mind the following when applying namespaces:

- You can apply only one namespace to each declaration.
- There is no way to apply a namespace attribute to more than one definition at a time. In other words, if you want to apply your namespace to ten different functions, you must add your namespace as an attribute to each of the ten function definitions.
- If you apply a namespace, you cannot also specify an access control specifier because namespaces and access control specifiers are mutually exclusive. In other words, you cannot declare a function or property as `public`, `private`, `protected`, or `internal` in addition to applying your namespace.

Referencing namespaces

There is no need to explicitly reference a namespace when you use a method or property declared with any of the access control namespaces, such as `public`, `private`, `protected`, and `internal`. This is because access to these special namespaces is controlled by context. For example, definitions placed into the `private` namespace are automatically available to code within the same class. For namespaces that you define, however, such context sensitivity does not exist. To use a method or property that you have placed into a custom namespace, you must reference the namespace.

You can reference namespaces with the `use namespace` directive or you can qualify the name with the namespace using the name qualifier (`::`) punctuator. Referencing a namespace with the `use namespace` directive "opens" the namespace, so that it can apply to any identifiers that are not qualified. For example, if you have defined the `example1` namespace, you can access names in that namespace by using `use namespace example1`:

```
use namespace example1;
myFunction();
```

You can open more than one namespace at a time. Once you open a namespace with `use namespace`, it remains open throughout the block of code in which it was opened. There is no way to explicitly close a namespace.

Having more than one open namespace, however, increases the likelihood of name conflicts. If you prefer not to open a namespace, you can avoid the `use namespace` directive by qualifying the method or property name with the namespace and the name qualifier punctuator. For example, the following code shows how you can qualify the name `myFunction()` with the `example1` namespace:

```
example1::myFunction();
```

Using namespaces

You can find a real-world example of a namespace that is used to prevent name conflicts in the flash.utils.Proxy class that is part of ActionScript 3.0. The Proxy class, which is the replacement for the `Object.__resolve` property from ActionScript 2.0, allows you to intercept references to undefined properties or methods before an error occurs. All of the methods of the Proxy class reside in the `flash_proxy` namespace to prevent name conflicts.

To better understand how the `flash_proxy` namespace is used, you need to understand how to use the Proxy class. The functionality of the Proxy class is available only to classes that inherit from it. In other words, if you want to use the methods of the Proxy class on an object, the object's class definition must extend the Proxy class. For example, if you want to intercept attempts to call an undefined method, you would extend the Proxy class and then override the `callProperty()` method of the Proxy class.

You may recall that implementing namespaces is usually a three-step process of defining, applying, and then referencing a namespace. Because you never explicitly call any of the Proxy class methods, however, the `flash_proxy` namespace is only defined and applied, but never referenced. ActionScript 3.0 defines the `flash_proxy` namespace and applies it in the Proxy class. Your code only needs to apply the `flash_proxy` namespace to classes that extend the Proxy class.

The `flash_proxy` namespace is defined in the flash.utils package in a manner similar to the following:

```
package flash.utils
{
    public namespace flash_proxy;
}
```

The namespace is applied to the methods of the Proxy class as shown in the following excerpt from the Proxy class:

```
public class Proxy
{
    flash_proxy function callProperty(name:*, ... rest):*
    flash_proxy function deleteProperty(name:*):Boolean
    ...
}
```

As the following code shows, you must first import both the Proxy class and the `flash_proxy` namespace. You must then declare your class such that it extends the Proxy class (you must also add the `dynamic` attribute if you are compiling in strict mode). When you override the `callProperty()` method, you must use the `flash_proxy` namespace.

```
package
{
    import flash.utils.Proxy;
    import flash.utils.flash_proxy;

    dynamic class MyProxy extends Proxy
    {
        flash_proxy override function callProperty(name:*, ...rest):*
        {
            trace("method call intercepted: " + name);
        }
    }
}
```

If you create an instance of the MyProxy class and call an undefined method, such as the `testing()` method called in the following example, your Proxy object intercepts the method call and executes the statements inside the overridden `callProperty()` method (in this case, a simple `trace()` statement).

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // method call intercepted: testing
```

There are two advantages to having the methods of the Proxy class inside the `flash_proxy` namespace. First, having a separate namespace reduces clutter in the public interface of any class that extends the Proxy class. (There are about a dozen methods in the Proxy class that you can override, all of which are not designed to be called directly. Placing all of them in the public namespace could be confusing.) Second, use of the `flash_proxy` namespace avoids name conflicts in case your Proxy subclass contains instance methods with names that match any of the Proxy class methods. For example, you may want to name one of your own methods `callProperty()`. The following code is acceptable, because your version of the `callProperty()` method is in a different namespace:

```
dynamic class MyProxy extends Proxy
{
    public function callProperty() {}
    flash_proxy override function callProperty(name:*, ...rest):*
    {
        trace("method call intercepted: " + name);
    }
}
```

Namespaces can also be helpful when you want to provide access to methods or properties in a way that cannot be accomplished with the four access control specifiers (`public`, `private`, `internal`, and `protected`). For example, you may have a few utility methods that are spread out across several packages. You want these methods available to all of your packages, but you don't want the methods to be public. To accomplish this, you can create a namespace and use it as your own special access control specifier.

The following example uses a user-defined namespace to group two functions that reside in different packages. By grouping them into the same namespace, you can make both functions visible to a class or package through a single `use namespace` statement.

This example uses four files to demonstrate the technique. All of the files must be within your classpath. The first file, myInternal.as, is used to define the `myInternal` namespace. Because the file is in a package named example, you must place the file into a folder named example. The namespace is marked as `public` so that it can be imported into other packages.

```
// myInternal.as in folder example
package example
{
    public namespace myInternal = "http://www.adobe.com/2006/actionscript/examples";
}
```

The second and third files, Utility.as and Helper.as, define the classes that contain methods that should be available to other packages. The Utility class is in the example.alpha package, which means that the file should be placed inside a folder named alpha that is a subfolder of the example folder. The Helper class is in the example.beta package, which means that the file should be placed inside a folder named beta that is also a subfolder of the example folder. Both of these packages, example.alpha and example.beta, must import the namespace before using it.

```
// Utility.as in the example/alpha folder
package example.alpha
{
    import example.myInternal;

    public class Utility
    {
        private static var _taskCounter:int = 0;

        public static function someTask()
        {
            _taskCounter++;
        }

        myInternal static function get taskCounter():int
        {
            return _taskCounter;
        }
    }
}
```

```
// Helper.as in the example/beta folder
package example.beta
{
    import example.myInternal;

    public class Helper
    {
        private static var _timeStamp:Date;

        public static function someTask()
        {
            _timeStamp = new Date();
        }

        myInternal static function get lastCalled():Date
        {
            return _timeStamp;
        }
    }
}
```

The fourth file, NamespaceUseCase.as, is the main application class, and should be a sibling to the example folder. In Flash Professional, this class would be used as the document class for the FLA. The NamespaceUseCase class also imports the `myInternal` namespace and uses it to call the two static methods that reside in the other packages. The example uses static methods only to simplify the code. Both static and instance methods can be placed in the `myInternal` namespace.

```
// NamespaceUseCase.as
package
{
    import flash.display.MovieClip;
    import example.myInternal; // import namespace
    import example.alpha.Utility;// import Utility class
    import example.beta.Helper;// import Helper class

    public class NamespaceUseCase extends MovieClip
    {
        public function NamespaceUseCase()
        {
            use namespace myInternal;

            Utility.someTask();
            Utility.someTask();
            trace(Utility.taskCounter); // 2

            Helper.someTask();
            trace(Helper.lastCalled); // [time someTask() was last called]
        }
    }
}
```

Variables

Variables allow you to store values that you use in your program. To declare a variable, you must use the `var` statement with the variable name. In ActionScript 3.0, use of the `var` statement is always required. For example, the following line of ActionScript declares a variable named `i`:

```
var i;
```

If you omit the `var` statement when declaring a variable, you get a compiler error in strict mode and run-time error in standard mode. For example, the following line of code results in an error if the variable `i` has not been previously defined:

```
i; // error if i was not previously defined
```

To associate a variable with a data type, you must do so when you declare the variable. Declaring a variable without designating the variable's type is legal, but generates a compiler warning in strict mode. You designate a variable's type by appending the variable name with a colon (:), followed by the variable's type. For example, the following code declares a variable `i` that is of type int:

```
var i:int;
```

You can assign a value to a variable using the assignment operator (`=`). For example, the following code declares a variable `i` and assigns the value 20 to it:

```
var i:int;
i = 20;
```

You may find it more convenient to assign a value to a variable at the same time that you declare the variable, as in the following example:

```
var i:int = 20;
```

The technique of assigning a value to a variable at the time it is declared is commonly used not only when assigning primitive values such as integers and strings, but also when creating an array or instantiating an instance of a class. The following example shows an array that is declared and assigned a value using one line of code.

```
var numArray:Array = ["zero", "one", "two"];
```

You can create an instance of a class by using the `new` operator. The following example creates an instance of a named `CustomClass`, and assigns a reference to the newly created class instance to the variable named `customItem`:

```
var customItem:CustomClass = new CustomClass();
```

If you have more than one variable to declare, you can declare them all on one line of code by using the comma operator (`,`) to separate the variables. For example, the following code declares three variables on one line of code:

```
var a:int, b:int, c:int;
```

You can also assign values to each of the variables on the same line of code. For example, the following code declares three variables (`a`, `b`, and `c`) and assigns each a value:

```
var a:int = 10, b:int = 20, c:int = 30;
```

Although you can use the comma operator to group variable declarations into one statement, doing so may reduce the readability of your code.

Understanding variable scope

The *scope* of a variable is the area of your code where the variable can be accessed by a lexical reference. A *global* variable is one that is defined in all areas of your code, whereas a *local* variable is one that is defined in only one part of your code. In ActionScript 3.0, variables are always assigned the scope of the function or class in which they are declared. A global variable is a variable that you define outside of any function or class definition. For example, the following code creates a global variable `strGlobal` by declaring it outside of any function. The example shows that a global variable is available both inside and outside the function definition.

```
var strGlobal:String = "Global";
function scopeTest()
{
    trace(strGlobal); // Global
}
scopeTest();
trace(strGlobal); // Global
```

You declare a local variable by declaring the variable inside a function definition. The smallest area of code for which you can define a local variable is a function definition. A local variable declared within a function exists only in that function. For example, if you declare a variable named `str2` within a function named `localScope()`, that variable is not available outside the function.

```
function localScope()
{
    var strLocal:String = "local";
}
localScope();
trace(strLocal); // error because strLocal is not defined globally
```

If the variable name you use for your local variable is already declared as a global variable, the local definition hides (or shadows) the global definition while the local variable is in scope. The global variable still exists outside of the function. For example, the following code creates a global string variable named `str1`, and then creates a local variable of the same name inside the `scopeTest()` function. The `trace` statement inside the function outputs the local value of the variable, but the `trace` statement outside the function outputs the global value of the variable.

```
var str1:String = "Global";
function scopeTest ()
{
    var str1:String = "Local";
    trace(str1); // Local
}
scopeTest();
trace(str1); // Global
```

ActionScript variables, unlike variables in C++ and Java, do not have block-level scope. A block of code is any group of statements between an opening curly bracket ({) and a closing curly bracket (}). In some programming languages, such as C++ and Java, variables declared inside a block of code are not available outside that block of code. This restriction of scope is called block-level scope, and does not exist in ActionScript. If you declare a variable inside a block of code, that variable is available not only in that block of code, but also in any other parts of the function to which the code block belongs. For example, the following function contains variables that are defined in various block scopes. All the variables are available throughout the function.

```
function blockTest (testArray:Array)
{
    var numElements:int = testArray.length;
    if (numElements > 0)
    {
        var elemStr:String = "Element #";
        for (var i:int = 0; i < numElements; i++)
        {
            var valueStr:String = i + ": " + testArray[i];
            trace(elemStr + valueStr);
        }
        trace(elemStr, valueStr, i); // all still defined
    }
    trace(elemStr, valueStr, i); // all defined if numElements > 0
}

blockTest(["Earth", "Moon", "Sun"]);
```

An interesting implication of the lack of block-level scope is that you can read or write to a variable before it is declared, as long as it is declared before the function ends. This is because of a technique called *hoisting*, which means that the compiler moves all variable declarations to the top of the function. For example, the following code compiles even though the initial `trace()` function for the `num` variable happens before the `num` variable is declared:

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

The compiler will not, however, hoist any assignment statements. This explains why the initial `trace()` of `num` results in `NaN` (not a number), which is the default value for variables of the Number data type. This means that you can assign values to variables even before they are declared, as shown in the following example:

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

Default values

A *default value* is the value that a variable contains before you set its value. You *initialize* a variable when you set its value for the first time. If you declare a variable, but do not set its value, that variable is *uninitialized*. The value of an uninitialized variable depends on its data type. The following table describes the default values of variables, organized by data type:

Data type	Default value
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
String	`null`

Data type	Default value
uint	0
Not declared (equivalent to type annotation `*`)	`undefined`
All other classes, including user-defined classes.	`null`

For variables of type Number, the default value is `NaN` (not a number), which is a special value defined by the IEEE-754 standard to mean a value that does not represent a number.

If you declare a variable, but do not declare its data type, the default data type `*` applies, which actually means that the variable is untyped. If you also do not initialize an untyped variable with a value, its default value is `undefined`.

For data types other than Boolean, Number, int, and uint, the default value of any uninitialized variable is `null`. This applies to all the classes defined by ActionScript 3.0, as well as any custom classes that you create.

The value `null` is not a valid value for variables of type Boolean, Number, int, or uint. If you attempt to assign a value of `null` to a such a variable, the value is converted to the default value for that data type. For variables of type Object, you can assign a value of `null`. If you attempt to assign the value `undefined` to a variable of type Object, the value is converted to `null`.

For variables of type Number, there is a special top-level function named `isNaN()` that returns the Boolean value `true` if the variable is not a number, and `false` otherwise.

Data types

A *data type* defines a set of values. For example, the Boolean data type is the set of exactly two values: `true` and `false`. In addition to the Boolean data type, ActionScript 3.0 defines several more commonly used data types, such as String, Number, and Array. You can define your own data types by using classes or interfaces to define a custom set of values. All values in ActionScript 3.0, whether they are primitive or complex, are objects.

A *primitive value* is a value that belongs to one of the following data types: Boolean, int, Number, String, and uint. Working with primitive values is usually faster than working with complex values, because ActionScript stores primitive values in a special way that makes memory and speed optimizations possible.

Note: *For readers interested in the technical details, ActionScript stores primitive values internally as immutable objects. The fact that they are stored as immutable objects means that passing by reference is effectively the same as passing by value. This cuts down on memory usage and increases execution speed, because references are usually significantly smaller than the values themselves.*

A *complex value* is a value that is not a primitive value. Data types that define sets of complex values include Array, Date, Error, Function, RegExp, XML, and XMLList.

Many programming languages distinguish between primitive values and their wrapper objects. Java, for example, has an int primitive and the java.lang.Integer class that wraps it. Java primitives are not objects, but their wrappers are, which makes primitives useful for some operations, and wrapper objects better suited for other operations. In ActionScript 3.0, primitive values and their wrapper objects are, for practical purposes, indistinguishable. All values, even primitive values, are objects. The runtime treats these primitive types as special cases that behave like objects but that don't require the normal overhead associated with creating objects. This means that the following two lines of code are equivalent:

```
var someInt:int = 3;
var someInt:int = new int(3);
```

All the primitive and complex data types listed above are defined by the ActionScript 3.0 core classes. The core classes allow you to create objects using literal values instead of using the `new` operator. For example, you can create an array using a literal value or the Array class constructor, as follows:

```
var someArray:Array = [1, 2, 3]; // literal value
var someArray:Array = new Array(1,2,3); // Array constructor
```

Type checking

Type checking can occur at either compile time or run time. Statically typed languages, such as C++ and Java, do type checking at compile time. Dynamically typed languages, such as Smalltalk and Python, handle type checking at run time. As a dynamically typed language, ActionScript 3.0 has run-time type checking, but also supports compile-time type checking with a special compiler mode called *strict mode*. In strict mode, type checking occurs at both compile time and run time, but in standard mode, type checking occurs only at run time.

Dynamically typed languages offer tremendous flexibility when you structure your code, but at the cost of allowing type errors to manifest at run time. Statically typed languages report type errors at compile time, but at the cost of requiring that type information be known at compile time.

Compile-time type checking

Compile-time type checking is often favored in larger projects because as the size of a project grows, data type flexibility usually becomes less important than catching type errors as early as possible. This is why, by default, the ActionScript compiler in Flash Professional and Flash Builder is set to run in strict mode.

Adobe Flash Builder

You can disable strict mode in Flash Builder through the ActionScript compiler settings in the Project Properties dialog box.

To provide compile-time type checking, the compiler needs to know the data type information for the variables or expressions in your code. To explicitly declare a data type for a variable, add the colon operator (:) followed by the data type as a suffix to the variable name. To associate a data type with a parameter, use the colon operator followed by the data type. For example, the following code adds data type information to the `xParam` parameter, and declares a variable `myParam` with an explicit data type:

```
function runtimeTest(xParam:String)
{
    trace(xParam);
}
var myParam:String = "hello";
runtimeTest(myParam);
```

In strict mode, the ActionScript compiler reports type mismatches as compiler errors. For example, the following code declares a function parameter `xParam`, of type Object, but later attempts to assign values of type String and Number to that parameter. This produces a compiler error in strict mode.

```
function dynamicTest(xParam:Object)
{
    if (xParam is String)
    {
        var myStr:String = xParam; // compiler error in strict mode
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam; // compiler error in strict mode
        trace("Number: " + myNum);
    }
}
```

Even in strict mode, however, you can selectively opt of out compile-time type checking by leaving the right side of an assignment statement untyped. You can mark a variable or expression as untyped by either omitting a type annotation, or using the special asterisk (`*`) type annotation. For example, if the `xParam` parameter in the previous example is modified so that it no longer has a type annotation, the code compiles in strict mode:

```
function dynamicTest(xParam)
{
    if (xParam is String)
    {
        var myStr:String = xParam;
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam;
        trace("Number: " + myNum);
    }
}
dynamicTest(100)
dynamicTest("one hundred");
```

Run-time type checking

Run-time type checking occurs in ActionScript 3.0 whether you compile in strict mode or standard mode. Consider a situation in which the value 3 is passed as an argument to a function that expects an array. In strict mode, the compiler will generate an error, because the value 3 is not compatible with the data type Array. If you disable strict mode, and run in standard mode, the compiler does not complain about the type mismatch, but run-time type checking results in a run-time error.

The following example shows a function named `typeTest()` that expects an Array argument but is passed a value of 3. This causes a run-time error in standard mode, because the value 3 is not a member of the parameter's declared data type (Array).

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0 standard mode
```

There may also be situations where you get a run-time type error even when you are operating in strict mode. This is possible if you use strict mode, but opt out of compile-time type checking by using an untyped variable. When you use an untyped variable, you are not eliminating type checking but rather deferring it until run time. For example, if the `myNum` variable in the previous example does not have a declared data type, the compiler cannot detect the type mismatch, but the code will generate a run-time error because it compares the run-time value of `myNum`, which is set to 3 as a result of the assignment statement, with the type of `xParam`, which is set to the Array data type.

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0
```

Run-time type checking also allows more flexible use of inheritance than does compile-time checking. By deferring type checking to run time, standard mode allows you to reference properties of a subclass even if you *upcast*. An upcast occurs when you use a base class to declare the type of a class instance but use a subclass to instantiate it. For example, you can create a class named ClassBase that can be extended (classes with the `final` attribute cannot be extended):

```
class ClassBase
{
}
```

You can subsequently create a subclass of ClassBase named ClassExtender, which has one property named `someString`, as follows:

```
class ClassExtender extends ClassBase
{
    var someString:String;
}
```

Using both classes, you can create a class instance that is declared using the ClassBase data type, but instantiated using the ClassExtender constructor. An upcast is considered a safe operation, because the base class does not contain any properties or methods that are not in the subclass.

```
var myClass:ClassBase = new ClassExtender();
```

A subclass, however, does contain properties or methods that its base class does not. For example, the ClassExtender class contains the `someString` property, which does not exist in the ClassBase class. In ActionScript 3.0 standard mode, you can reference this property using the `myClass` instance without generating a compile-time error, as shown in the following example:

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// no error in ActionScript 3.0 standard mode
```

The is operator

The `is` operator allows you to test whether a variable or expression is a member of a given data type. In previous versions of ActionScript, the `instanceof` operator provided this functionality, but in ActionScript 3.0 the `instanceof` operator should not be used to test for data type membership. The `is` operator should be used instead of the `instanceof` operator for manual type checking, because the expression `x instanceof y` merely checks the prototype chain of `x` for the existence of `y` (and in ActionScript 3.0, the prototype chain does not provide a complete picture of the inheritance hierarchy).

The `is` operator examines the proper inheritance hierarchy and can be used to check not only whether an object is an instance of a particular class, but also whether an object is an instance of a class that implements a particular interface. The following example creates an instance of the Sprite class named `mySprite` and uses the `is` operator to test whether `mySprite` is an instance of the Sprite and DisplayObject classes, and whether it implements the IEventDispatcher interface:

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite); // true
trace(mySprite is DisplayObject);// true
trace(mySprite is IEventDispatcher); // true
```

The `is` operator checks the inheritance hierarchy and properly reports that `mySprite` is compatible with the Sprite and DisplayObject classes (the Sprite class is a subclass of the DisplayObject class). The `is` operator also checks whether `mySprite` inherits from any classes that implement the IEventDispatcher interface. Because the Sprite class inherits from the EventDispatcher class, which implements the IEventDispatcher interface, the `is` operator correctly reports that `mySprite` implements the same interface.

The following example shows the same tests from the previous example, but with `instanceof` instead of the `is` operator. The `instanceof` operator correctly identifies that `mySprite` is an instance of Sprite or DisplayObject, but it returns `false` when used to test whether `mySprite` implements the IEventDispatcher interface.

```
trace(mySprite instanceof Sprite); // true
trace(mySprite instanceof DisplayObject);// true
trace(mySprite instanceof IEventDispatcher); // false
```

The as operator

The `as` operator also allows you to check whether an expression is a member of a given data type. Unlike the `is` operator, however, the `as` operator does not return a Boolean value. Rather, the `as` operator returns the value of the expression instead of `true`, and `null` instead of `false`. The following example shows the results of using the `as` operator instead of the `is` operator in the simple case of checking whether a Sprite instance is a member of the DisplayObject, IEventDispatcher, and Number data types.

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [object Sprite]
trace(mySprite as DisplayObject); // [object Sprite]
trace(mySprite as IEventDispatcher); // [object Sprite]
trace(mySprite as Number);          // null
```

When you use the `as` operator, the operand on the right must be a data type. An attempt to use an expression other than a data type as the operand on the right will result in an error.

Dynamic classes

A *dynamic* class defines an object that can be altered at run time by adding or changing properties and methods. A class that is not dynamic, such as the String class, is a *sealed* class. You cannot add properties or methods to a sealed class at run time.

You create dynamic classes by using the `dynamic` attribute when you declare a class. For example, the following code creates a dynamic class named `Protean`:

```
dynamic class Protean
{
    private var privateGreeting:String = "hi";
    public var publicGreeting:String = "hello";
    function Protean()
    {
        trace("Protean instance created");
    }
}
```

If you later instantiate an instance of the `Protean` class, you can add properties or methods to it outside the class definition. For example, the following code creates an instance of the `Protean` class and adds a property named `aString` and a property named `aNumber` to the instance:

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // testing 3
```

Properties that you add to an instance of a dynamic class are run-time entities, so any type checking is done at run time. You cannot add a type annotation to a property that you add in this manner.

You can also add a method to the `myProtean` instance by defining a function and attaching the function to a property of the `myProtean` instance. The following code moves the trace statement into a method named `traceProtean()`:

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
    trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // testing 3
```

Methods created in this way, however, do not have access to any private properties or methods of the Protean class. Moreover, even references to public properties or methods of the `Protean` class must be qualified with either the `this` keyword or the class name. The following example shows the `traceProtean()` method attempting to access the private and public variables of the `Protean` class.

```
myProtean.traceProtean = function ()
{
    trace(myProtean.privateGreeting); // undefined
    trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

Data type descriptions

The primitive data types include Boolean, int, Null, Number, String, uint, and void. The ActionScript core classes also define the following complex data types: Object, Array, Date, Error, Function, RegExp, XML, and XMLList.

Boolean data type

The Boolean data type includes two values: `true` and `false`. No other values are valid for variables of Boolean type. The default value of a Boolean variable that has been declared but not initialized is `false`.

int data type

The int data type is stored internally as a 32-bit integer and includes the set of integers from

-2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$), inclusive. Previous versions of ActionScript offered only the Number data type, which was used for both integers and floating-point numbers. In ActionScript 3.0, you now have access to low-level machine types for 32-bit signed and unsigned integers. If your variable does not need floating-point numbers, using the int data type instead of the Number data type is faster and more efficient.

For integer values outside the range of the minimum and maximum int values, use the Number data type, which can handle values between positive and negative 9,007,199,254,740,992 (53-bit integer values). The default value for variables that are of the data type int is 0.

Null data type

The Null data type contains only one value, `null`. This is the default value for the String data type and all classes that define complex data types, including the Object class. None of the other primitive data types, such as Boolean, Number, int, and uint, contain the value `null`. At run time, the value `null` is converted to the appropriate default value if you attempt to assign `null` to variables of type Boolean, Number, int, or uint. You cannot use this data type as a type annotation.

Number data type

In ActionScript 3.0, the Number data type can represent integers, unsigned integers, and floating-point numbers. However, to maximize performance, you should use the Number data type only for integer values larger than the 32-bit `int` and `uint` types can store or for floating-point numbers. To store a floating-point number, include a decimal point in the number. If you omit a decimal point, the number is stored as an integer.

The Number data type uses the 64-bit double-precision format as specified by the IEEE Standard for Binary Floating-Point Arithmetic (IEEE-754). This standard dictates how floating-point numbers are stored using the 64 available bits. One bit is used to designate whether the number is positive or negative. Eleven bits are used for the exponent, which is stored as base 2. The remaining 52 bits are used to store the *significand* (also called the *mantissa*), which is the number that is raised to the power indicated by the exponent.

By using some of its bits to store an exponent, the Number data type can store floating-point numbers significantly larger than if it used all of its bits for the significand. For example, if the Number data type used all 64 bits to store the significand, it could store a number as large as $2^{65} - 1$. By using 11 bits to store an exponent, the Number data type can raise its significand to a power of 2^{1023}.

The maximum and minimum values that the Number type can represent are stored in static properties of the Number class called `Number.MAX_VALUE` and `Number.MIN_VALUE`.

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

Although this range of numbers is enormous, the cost of this range is precision. The Number data type uses 52 bits to store the significand, with the result that numbers that require more than 52 bits to represent precisely, such as the fraction 1/3, are only approximations. If your application requires absolute precision with decimal numbers, you need to use software that implements decimal floating-point arithmetic as opposed to binary floating-point arithmetic.

When you store integer values with the Number data type, only the 52 bits of the significand are used. The Number data type uses these 52 bits and a special hidden bit to represent integers from -9,007,199,254,740,992 (-2^{53}) to 9,007,199,254,740,992 (2^{53}).

The `NaN` value is not only used as the default value for variables of type `Number`, but also as the result of any operation that should return a number but does not. For example, if you attempt to calculate the square root of a negative number, the result is `NaN`. Other special Number values include *positive infinity* and *negative infinity*.

Note: *The result of division by* `0` *is only* `NaN` *if the divisor is also* `0`*. Division by* `0` *produces* `infinity` *when the dividend is positive or* `-infinity` *when the dividend is negative.*

String data type

The String data type represents a sequence of 16-bit characters. Strings are stored internally as Unicode characters, using the UTF-16 format. Strings are immutable values, just as they are in the Java programming language. An operation on a String value returns a new instance of the string. The default value for a variable declared with the String data type is `null`. The value `null` is not the same as the empty string (`""`). The value `null` means that the variable has no value stored in it, while the empty string means that the variable has a value that is a String containing no characters.

uint data type

The uint data type is stored internally as a 32-bit unsigned integer and includes the set of integers from 0 to 4,294,967,295 (2^{32} - 1), inclusive. Use the uint data type for special circumstances that call for non-negative integers. For example, you must use the uint data type to represent pixel color values, because the int data type has an internal sign bit that is not appropriate for handling color values. For integer values larger than the maximum uint value, use the Number data type, which can handle 53-bit integer values. The default value for variables that are of data type uint is 0.

void data type

The void data type contains only one value, `undefined`. In previous versions of ActionScript, `undefined` was the default value for instances of the Object class. In ActionScript 3.0, the default value for Object instances is `null`. If you attempt to assign the value `undefined` to an instance of the Object class, the value is converted to `null`. You can only assign a value of `undefined` to variables that are untyped. Untyped variables are variables that either lack any type annotation, or use the asterisk (`*`) symbol for type annotation. You can use `void` only as a return type annotation.

Object data type

The Object data type is defined by the Object class. The Object class serves as the base class for all class definitions in ActionScript. The ActionScript 3.0 version of the Object data type differs from that of previous versions in three ways. First, the Object data type is no longer the default data type assigned to variables with no type annotation. Second, the Object data type no longer includes the value `undefined`, which used to be the default value of Object instances. Third, in ActionScript 3.0, the default value for instances of the Object class is `null`.

In previous versions of ActionScript, a variable with no type annotation was automatically assigned the Object data type. This is no longer true in ActionScript 3.0, which now includes the idea of a truly untyped variable. Variables with no type annotation are now considered untyped. If you prefer to make it clear to readers of your code that your intention is to leave a variable untyped, you can use the asterisk (`*`) symbol for the type annotation, which is equivalent to omitting a type annotation. The following example shows two equivalent statements, both of which declare an untyped variable `x`:

```
var x
var x:*
```

Only untyped variables can hold the value `undefined`. If you attempt to assign the value `undefined` to a variable that has a data type, the runtime converts the value `undefined` to the default value of that data type. For instances of the Object data type, the default value is `null`, which means that if you attempt to assign `undefined` to an Object instance the value is converted to `null`.

Type conversions

A type conversion is said to occur when a value is transformed into a value of a different data type. Type conversions can be either *implicit* or *explicit*. Implicit conversion, which is also called *coercion*, is sometimes performed at run time. For example, if the value 2 is assigned to a variable of the Boolean data type, the value 2 is converted to the Boolean value `true` before assigning the value to the variable. Explicit conversion, which is also called *casting*, occurs when your code instructs the compiler to treat a variable of one data type as if it belongs to a different data type. When primitive values are involved, casting actually converts values from one data type to another. To cast an object to a different type, you wrap the object name in parentheses and precede it with the name of the new type. For example, the following code takes a Boolean value and casts it to an integer:

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

Implicit conversions

Implicit conversions happen at run time in a number of contexts:

- In assignment statements
- When values are passed as function arguments
- When values are returned from functions
- In expressions using certain operators, such as the addition (+) operator

  For user-defined types, implicit conversions succeed when the value to be converted is an instance of the destination class or a class that derives from the destination class. If an implicit conversion is unsuccessful, an error occurs. For example, the following code contains a successful implicit conversion and an unsuccessful implicit conversion:

  ```
  class A {}
  class B extends A {}

  var objA:A = new A();
  var objB:B = new B();
  var arr:Array = new Array();

  objA = objB; // Conversion succeeds.
  objB = arr; // Conversion fails.
  ```

  For primitive types, implicit conversions are handled by calling the same internal conversion algorithms that are called by the explicit conversion functions.

Explicit conversions

It's helpful to use explicit conversions, or casting, when you compile in strict mode, because there may be times when you do not want a type mismatch to generate a compile-time error. This may be the case when you know that coercion will convert your values correctly at run time. For example, when working with data received from a form, you may want to rely on coercion to convert certain string values to numeric values. The following code generates a compile-time error even though the code would run correctly in standard mode:

```
var quantityField:String = "3";
var quantity:int = quantityField; // compile time error in strict mode
```

If you want to continue using strict mode, but would like the string converted to an integer, you can use explicit conversion, as follows:

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // Explicit conversion succeeds.
```

Casting to int, uint, and Number

You can cast any data type into one of the three number types: int, uint, and Number. If the number can't be converted for some reason, the default value of 0 is assigned for the int and uint data types, and the default value of `NaN` is assigned for the Number data type. If you convert a Boolean value to a number, `true` becomes the value 1 and `false` becomes the value 0.

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

String values that contain only digits can be successfully converted into one of the number types. The number types can also convert strings that look like negative numbers or strings that represent a hexadecimal value (for example, `0x1A`). The conversion process ignores leading and trailing white space characters in the string value. You can also cast strings that look like floating-point numbers using `Number()`. The inclusion of a decimal point causes `uint()` and `int()` to return an integer, truncating the decimal and the characters following it. For example, the following string values can be cast into numbers:

```
trace(uint("5")); // 5
trace(uint("-5")); // 4294967291. It wraps around from MAX_VALUE
trace(uint(" 27 ")); // 27
trace(uint("3.7")); // 3
trace(int("3.7")); // 3
trace(int("0x1A")); // 26
trace(Number("3.7")); // 3.7
```

String values that contain non-numeric characters return 0 when cast with `int()` or `uint()` and `NaN` when cast with `Number()`. The conversion process ignores leading and trailing white space, but returns 0 or `NaN` if a string has white space separating two numbers.

```
trace(uint("5a")); // 0
trace(uint("ten")); // 0
trace(uint("17 63")); // 0
```

In ActionScript 3.0, the `Number()` function no longer supports octal, or base 8, numbers. If you supply a string with a leading zero to the ActionScript 2.0 `Number()` function, the number is interpreted as an octal number, and converted to its decimal equivalent. This is not true with the `Number()` function in ActionScript 3.0, which instead ignores the leading zero. For example, the following code generates different output when compiled using different versions of ActionScript:

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

Casting is not necessary when a value of one numeric type is assigned to a variable of a different numeric type. Even in strict mode, the numeric types are implicitly converted to the other numeric types. This means that in some cases, unexpected values may result when the range of a type is exceeded. The following examples all compile in strict mode, though some generate unexpected values:

```
var myUInt:uint = -3; // Assign int/Number value to uint variable
trace(myUInt); // 4294967293

var myNum:Number = sampleUINT; // Assign int/uint value to Number variable
trace(myNum) // 4294967293

var myInt:int = uint.MAX_VALUE + 1; // Assign Number value to uint variable
trace(myInt); // 0

myInt = int.MAX_VALUE + 1; // Assign uint/Number value to int variable
trace(myInt); // -2147483648
```

The following table summarizes the results of casting to the Number, int, or uint data type from other data types.

Data type or value	Result of conversion to Number, int, or uint
Boolean	If the value is `true`, 1; otherwise, 0.
Date	The internal representation of the Date object, which is the number of milliseconds since midnight January 1, 1970, universal time.
`null`	0
Object	If the instance is `null` and converted to Number, `NaN`; otherwise, 0.
String	A number if the string can be converted to a number; otherwise, `NaN` if converted to Number, or 0 if converted to int or uint.
`undefined`	If converted to Number, `NaN`; if converted to int or uint, 0.

Casting to Boolean

Casting to Boolean from any of the numeric data types (uint, int, and Number) results in `false` if the numeric value is 0, and `true` otherwise. For the Number data type, the value `NaN` also results in `false`. The following example shows the results of casting the numbers -1, 0, and 1:

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
    trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

The output from the example shows that, of the three numbers, only 0 returns a value of `false`:

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

Casting to Boolean from a String value returns `false` if the string is either `null` or an empty string (`""`). Otherwise, it returns `true`.

```
var str1:String; // Uninitialized string is null.
trace(Boolean(str1)); // false

var str2:String = ""; // empty string
trace(Boolean(str2)); // false

var str3:String = " "; // white space only
trace(Boolean(str3)); // true
```

Casting to Boolean from an instance of the Object class returns `false` if the instance is `null`; otherwise, it returns `true`:

```
var myObj:Object; // Uninitialized object is null.
trace(Boolean(myObj)); // false

myObj = new Object(); // instantiate
trace(Boolean(myObj)); // true
```

Boolean variables get special treatment in strict mode in that you can assign values of any data type to a Boolean variable without casting. Implicit coercion from all data types to the Boolean data type occurs even in strict mode. In other words, unlike almost all other data types, casting to Boolean is not necessary to avoid strict mode errors. The following examples all compile in strict mode and behave as expected at run time:

```
var myObj:Object = new Object(); // instantiate
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

The following table summarizes the results of casting to the Boolean data type from other data types:

Data type or value	Result of conversion to Boolean
String	`false` if the value is `null` or the empty string (`""`); `true` otherwise.
`null`	`false`
Number, int, or uint	`false` if the value is `NaN` or 0; `true` otherwise.
Object	`false` if the instance is `null`; `true` otherwise.

Casting to String

Casting to the String data type from any of the numeric data types returns a string representation of the number. Casting to the String data type from a Boolean value returns the string `"true"` if the value is `true`, and returns the string `"false"` if the value is `false`.

Casting to the String data type from an instance of the Object class returns the string `"null"` if the instance is `null`. Otherwise, casting to the String type from the Object class returns the string `"[object Object]"` .

Casting to String from an instance of the Array class returns a string made up of a comma-delimited list of all the array elements. For example, the following cast to the String data type returns one string containing all three elements of the array:

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

Casting to String from an instance of the Date class returns a string representation of the date that the instance contains. For example, the following example returns a string representation of the Date class instance (the output shows result for Pacific Daylight Time):

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // Fri Jul 1 00:00:00 GMT-0700 2005
```

The following table summarizes the results of casting to the String data type from other data types.

Data type or value	Result of conversion to string
Array	A string made up of all array elements.
Boolean	`"true"` or `"false"`
Date	A string representation of the Date object.
`null`	`"null"`
Number, int, or uint	A string representation of the number.
Object	If the instance is null, `"null"`; otherwise, `"[object Object]"`.

Syntax

The syntax of a language defines a set of rules that must be followed when writing executable code.

Case sensitivity

ActionScript 3.0 is a case-sensitive language. Identifiers that differ only in case are considered different identifiers. For example, the following code creates two different variables:

```
var num1:int;
var Num1:int;
```

Dot syntax

The dot operator (.) provides a way to access the properties and methods of an object. Using dot syntax, you can refer to a class property or method by using an instance name, followed by the dot operator and name of the property or method. For example, consider the following class definition:

```
class DotExample
{
    public var prop1:String;
    public function method1():void {}
}
```

Using dot syntax, you can access the `prop1` property and the `method1()` method by using the instance name created in the following code:

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = "hello";
myDotEx.method1();
```

You can use dot syntax when you define packages. You use the dot operator to refer to nested packages. For example, the EventDispatcher class resides in a package named events that is nested within the package named flash. You can refer to the events package using the following expression:

```
flash.events
```

You can also refer to the EventDispatcher class using this expression:

```
flash.events.EventDispatcher
```

Slash syntax

Slash syntax is not supported in ActionScript 3.0. Slash syntax was used in earlier versions of ActionScript to indicate the path of a movie clip or variable.

Literals

A *literal* is a value that appears directly in your code. The following examples are all literals:

```
17
"hello"
-3
9.4
null
undefined
true
false
```

Literals can also be grouped to form compound literals. Array literals are enclosed in bracket characters (`[]`) and use the comma to separate array elements.

An array literal can be used to initialize an array. The following examples show two arrays that are initialized using array literals. You can use the `new` statement and pass the compound literal as a parameter to the Array class constructor, but you can also assign literal values directly when instantiating instances of the following ActionScript core classes: Object, Array, String, Number, int, uint, XML, XMLList, and Boolean.

```
// Use new statement.
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// Assign literal directly.
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

Literals can also be used to initialize a generic object. A generic object is an instance of the Object class. Object literals are enclosed in curly brackets ({}) and use the comma to separate object properties. Each property is declared with the colon character (`:`), which separates the name of the property from the value of the property.

You can create a generic object using the `new` statement, and pass the object literal as a parameter to the Object class constructor, or you can assign the object literal directly to the instance you are declaring. The following example demonstrates two alternative ways to create a new generic object and initialize the object with three properties (`propA`, `propB`, and `propC`), each with values set to 1, 2, and 3, respectively:

```
// Use new statement and add properties.
var myObject:Object = new Object();
myObject.propA = 1;
myObject.propB = 2;
myObject.propC = 3;

// Assign literal directly.
var myObject:Object = {propA:1, propB:2, propC:3};
```

More Help topics

Working with strings

Using regular expressions

Initializing XML variables

Semicolons

You can use the semicolon character (`;`) to terminate a statement. Alternatively, if you omit the semicolon character, the compiler assumes that each line of code represents a single statement. Because many programmers are accustomed to using the semicolon to denote the end of a statement, your code may be easier to read if you consistently use semicolons to terminate your statements.

Using a semicolon to terminate a statement allows you to place more than one statement on a single line, but this may make your code more difficult to read.

Parentheses

You can use parentheses (`()`) in three ways in ActionScript 3.0. First, you can use parentheses to change the order of operations in an expression. Operations that are grouped inside parentheses are always executed first. For example, parentheses are used to alter the order of operations in the following code:

```
trace(2 + 3 * 4); // 14
trace((2 + 3) * 4); // 20
```

Second, you can use parentheses with the comma operator (`,`) to evaluate a series of expressions and return the result of the final expression, as shown in the following example:

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

Third, you can use parentheses to pass one or more parameters to functions or methods, as shown in the following example, which passes a String value to the `trace()` function:

```
trace("hello"); // hello
```

Comments

ActionScript 3.0 code supports two types of comments: single-line comments and multiline comments. These commenting mechanisms are similar to the commenting mechanisms in C++ and Java. The compiler ignores text that is marked as a comment.

Single-line comments begin with two forward slash characters (`//`) and continue until the end of the line. For example, the following code contains a single-line comment:

```
var someNumber:Number = 3; // a single line comment
```

Multiline comments begin with a forward slash and asterisk (`/*`) and end with an asterisk and forward slash (`*/`).

```
/* This is multiline comment that can span
more than one line of code. */
```

Keywords and reserved words

Reserved words are words that you cannot use as identifiers in your code because the words are reserved for use by ActionScript. Reserved words include *lexical keywords*, which are removed from the program namespace by the compiler. The compiler reports an error if you use a lexical keyword as an identifier. The following table lists ActionScript 3.0 lexical keywords.

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	function
if	implements	import	in
instanceof	interface	internal	is
native	new	null	package
private	protected	public	return
super	switch	this	throw
to	true	try	typeof
use	var	void	while
with			

There is a small set of keywords, called *syntactic keywords*, that can be used as identifiers, but that have special meaning in certain contexts. The following table lists ActionScript 3.0 syntactic keywords.

each	get	set	namespace
include	dynamic	final	native
override	static		

There are also several identifiers that are sometimes referred to as *future reserved words*. These identifiers are not reserved by ActionScript 3.0, though some of them may be treated as keywords by software that incorporates ActionScript 3.0. You might be able to use many of these identifiers in your code, but Adobe recommends that you do not use them because they may appear as keywords in a subsequent version of the language.

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic
long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

Constants

ActionScript 3.0 supports the `const` statement, which you can use to create constants. Constants are properties with a fixed value that cannot be altered. You can assign a value to a constant only once, and the assignment must occur in close proximity to the declaration of the constant. For example, if a constant is declared as a member of a class, you can assign a value to that constant only as part of the declaration or inside the class constructor.

The following code declares two constants. The first constant, `MINIMUM`, has a value assigned as part of the declaration statement. The second constant, `MAXIMUM`, has a value assigned in the constructor. Note that this example only compiles in standard mode because strict mode only allows a constant's value to be assigned at initialization time.

```
class A
{
    public const MINIMUM:int = 0;
    public const MAXIMUM:int;

    public function A()
    {
        MAXIMUM = 10;
    }
}

var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

An error results if you attempt to assign an initial value to a constant in any other way. For example, if you attempt to set the initial value of `MAXIMUM` outside the class, a run-time error occurs.

```
class A
{
public const MINIMUM:int = 0;
public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // run-time error
```

ActionScript 3.0 defines a wide range of constants for your use. By convention, constants in ActionScript use all capital letters, with words separated by the underscore character (_). For example, the MouseEvent class definition uses this naming convention for its constants, each of which represents an event related to mouse input:

```
package flash.events
{
    public class MouseEvent extends Event
    {
    public static const CLICK:String = "click";
    public static const DOUBLE_CLICK:String = "doubleClick";
    public static const MOUSE_DOWN:String = "mouseDown";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...
    }
}
```

Operators

Operators are special functions that take one or more operands and return a value. An *operand* is a value—usually a literal, a variable, or an expression—that an operator uses as input. For example, in the following code, the addition (`+`) and multiplication (`*`) operators are used with three literal operands (`2`, `3`, and `4`) to return a value. This value is then used by the assignment (`=`) operator to assign the returned value, 14, to the variable `sumNumber`.

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

Operators can be unary, binary, or ternary. A *unary* operator takes one operand. For example, the increment (`++`) operator is a unary operator, because it takes only one operand. A *binary* operator takes two operands. For example, the division (`/`) operator takes two operands. A *ternary* operator takes three operands. For example, the conditional (`?:`) operator takes three operands.

Some operators are *overloaded*, which means that they behave differently depending on the type or quantity of operands passed to them. The addition (`+`) operator is an example of an overloaded operator that behaves differently depending on the data type of the operands. If both operands are numbers, the addition operator returns the sum of the values. If both operands are strings, the addition operator returns the concatenation of the two operands. The following example code shows how the operator behaves differently depending on the operands:

```
trace(5 + 5); // 10
trace("5" + "5"); // 55
```

Operators can also behave differently based on the number of operands supplied. The subtraction (`-`) operator is both a unary and binary operator. When supplied with only one operand, the subtraction operator negates the operand and returns the result. When supplied with two operands, the subtraction operator returns the difference between the operands. The following example shows the subtraction operator used first as a unary operator, and then as a binary operator.

```
trace(-3); // -3
trace(7 - 2); // 5
```

Operator precedence and associativity

Operator precedence and associativity determine the order in which operators are processed. Although it may seem natural to those familiar with arithmetic that the compiler processes the multiplication (`*`) operator before the addition (`+`) operator, the compiler needs explicit instructions about which operators to process first. Such instructions are collectively referred to as *operator precedence*. ActionScript defines a default operator precedence that you can alter using the parentheses (`()`) operator. For example, the following code alters the default precedence in the previous example to force the compiler to process the addition operator before the multiplication operator:

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

You may encounter situations in which two or more operators of the same precedence appear in the same expression. In these cases, the compiler uses the rules of *associativity* to determine which operator to process first. All of the binary operators, except the assignment operators, are *left-associative*, which means that operators on the left are processed before operators on the right. The assignment operators and the conditional (`?:`) operator are *right-associative*, which means that the operators on the right are processed before operators on the left.

For example, consider the less-than (`<`) and greater-than (`>`) operators, which have the same precedence. If both operators are used in the same expression, the operator on the left is processed first because both operators are left-associative. This means that the following two statements produce the same output:

```
trace(3 > 2 < 1); // false
trace((3 > 2) < 1); // false
```

The greater-than operator is processed first, which results in a value of `true`, because the operand 3 is greater than the operand 2. The value `true` is then passed to the less-than operator along with the operand 1. The following code represents this intermediate state:

```
trace((true) < 1);
```

The less-than operator converts the value `true` to the numeric value 1 and compares that numeric value to the second operand 1 to return the value `false` (the value 1 is not less than 1).

```
trace(1 < 1); // false
```

You can alter the default left associativity with the parentheses operator. You can instruct the compiler to process the less-than operator first by enclosing that operator and its operands in parentheses. The following example uses the parentheses operator to produce a different output using the same numbers as the previous example:

```
trace(3 > (2 < 1)); // true
```

The less-than operator is processed first, which results in a value of `false` because the operand 2 is not less than the operand 1. The value `false` is then passed to the greater-than operator along with the operand 3. The following code represents this intermediate state:

```
trace(3 > (false));
```

The greater-than operator converts the value `false` to the numeric value 0 and compares that numeric value to the other operand 3 to return `true` (the value 3 is greater than 0).

```
trace(3 > 0); // true
```

The following table lists the operators for ActionScript 3.0 in order of decreasing precedence. Each row of the table contains operators of the same precedence. Each row of operators has higher precedence than the row appearing below it in the table.

Group	**Operators**
Primary	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Postfix	`x++ x--`
Unary	`++x --x + - ~ ! delete typeof void`
Multiplicative	`* / %`
Additive	`+ -`
Bitwise shift	`<< >> >>>`
Relational	`< > <= >= as in instanceof is`
Equality	`== != === !==`
Bitwise AND	`&`
Bitwise XOR	`^`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Conditional	`?:`
Assignment	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Comma	`,`

Primary operators

The primary operators include those used for creating Array and Object literals, grouping expressions, calling functions, instantiating class instances, and accessing properties.

All the primary operators, as listed in the following table, have equal precedence. Operators that are part of the E4X specification are indicated by the (E4X) notation.

Operator	Operation performed
`[]`	Initializes an array
`{x:y}`	Initializes an object
`()`	Groups expressions
`f(x)`	Calls a function
`new`	Calls a constructor
`x.y x[y]`	Accesses a property
`<></>`	Initializes an XMLList object (E4X)
`@`	Accesses an attribute (E4X)
`::`	Qualifies a name (E4X)
`..`	Accesses a descendant XML element (E4X)

Postfix operators

The postfix operators take one operator and either increment or decrement the value. Although these operators are unary operators, they are classified separately from the rest of the unary operators because of their higher precedence and special behavior. When a postfix operator is used as part of a larger expression, the expression's value is returned before the postfix operator is processed. For example, the following code shows how the value of the expression `xNum++` is returned before the value is incremented:

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum); // 1
```

All the postfix operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`++`	Increments (postfix)
`--`	Decrements (postfix)

Unary operators

The unary operators take one operand. The increment (`++`) and decrement (`--`) operators in this group are *prefixoperators*, which means that they appear before the operand in an expression. The prefix operators differ from their postfix counterparts in that the increment or decrement operation is completed before the value of the overall expression is returned. For example, the following code shows how the value of the expression `++xNum` is returned after the value is incremented:

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum); // 1
```

All the unary operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`++`	Increments (prefix)
`--`	Decrements (prefix)
`+`	Unary +
`-`	Unary - (negation)
`!`	Logical NOT
`~`	Bitwise NOT
`delete`	Deletes a property
`typeof`	Returns type information
`void`	Returns undefined value

Multiplicative operators

The multiplicative operators take two operands and perform multiplication, division, or modulo calculations.

All the multiplicative operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`*`	Multiplication
`/`	Division
`%`	Modulo

Additive operators

The additive operators take two operands and perform addition or subtraction calculations. All the additive operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`+`	Addition
`-`	Subtraction

Bitwise shift operators

The bitwise shift operators take two operands and shift the bits of the first operand to the extent specified by the second operand. All the bitwise shift operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`<<`	Bitwise left shift
`>>`	Bitwise right shift
`>>>`	Bitwise unsigned right shift

Relational operators

The relational operators take two operands, compare their values, and return a Boolean value. All the relational operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`as`	Checks data type
`in`	Checks for object properties
`instanceof`	Checks prototype chain
`is`	Checks data type

Equality operators

The equality operators take two operands, compare their values, and return a Boolean value. All the equality operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`==`	Equality
`!=`	Inequality
`===`	Strict equality
`!==`	Strict inequality

Bitwise logical operators

The bitwise logical operators take two operands and perform bit-level logical operations. The bitwise logical operators differ in precedence and are listed in the following table in order of decreasing precedence:

Operator	Operation performed
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR

Logical operators

The logical operators take two operands and return a Boolean result. The logical operators differ in precedence and are listed in the following table in order of decreasing precedence:

Operator	Operation performed
`&&`	Logical AND
`||`	Logical OR

Conditional operator

The conditional operator is a ternary operator, which means that it takes three operands. The conditional operator is a shorthand method of applying the `if..else` conditional statement.

Operator	Operation performed
`?:`	Conditional

Assignment operators

The assignment operators take two operands and assign a value to one operand, based on the value of the other operand. All the assignment operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`=`	Assignment
`*=`	Multiplication assignment
`/=`	Division assignment
`%=`	Modulo assignment
`+=`	Addition assignment
`-=`	Subtraction assignment
`<<=`	Bitwise left shift assignment
`>>=`	Bitwise right shift assignment
`>>>=`	Bitwise unsigned right shift assignment
`&=`	Bitwise AND assignment
`^=`	Bitwise XOR assignment
`|=`	Bitwise OR assignment

Conditionals

ActionScript 3.0 provides three basic conditional statements that you can use to control program flow.

if..else

The `if..else` conditional statement allows you to test a condition and execute a block of code if that condition exists, or execute an alternative block of code if the condition does not exist. For example, the following code tests whether the value of `x` exceeds 20, generates a `trace()` function if it does, or generates a different `trace()` function if it does not:

```
if (x > 20)
{
    trace("x is > 20");
}
else
{
    trace("x is <= 20");
}
```

If you do not want to execute an alternative block of code, you can use the `if` statement without the `else` statement.

if..else if

You can test for more than one condition using the `if..else if` conditional statement. For example, the following code not only tests whether the value of `x` exceeds 20, but also tests whether the value of `x` is negative:

```
if (x > 20)
{
    trace("x is > 20");
}
else if (x < 0)
{
    trace("x is negative");
}
```

If an `if` or `else` statement is followed by only one statement, the statement does not need to be enclosed in curly brackets. For example, the following code does not use curly brackets:

```
if (x > 0)
    trace("x is positive");
else if (x < 0)
    trace("x is negative");
else
    trace("x is 0");
```

However, Adobe recommends that you always use curly brackets, because unexpected behavior can occur if statements are later added to a conditional statement that lacks curly brackets. For example, in the following code the value of `positiveNums` increases by 1 whether or not the condition evaluates to `true`:

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
    trace("x is positive");
    positiveNums++;

trace(positiveNums); // 1
```

switch

The `switch` statement is useful if you have several execution paths that depend on the same condition expression. It provides functionality similar to a long series of `if..else if` statements, but is somewhat easier to read. Instead of testing a condition for a Boolean value, the `switch` statement evaluates an expression and uses the result to determine which block of code to execute. Blocks of code begin with a `case` statement and end with a `break` statement. For example, the following `switch` statement prints the day of the week, based on the day number returned by the `Date.getDay()` method:

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
    case 0:
        trace("Sunday");
        break;
    case 1:
        trace("Monday");
        break;
    case 2:
        trace("Tuesday");
        break;
    case 3:
        trace("Wednesday");
        break;
    case 4:
        trace("Thursday");
        break;
    case 5:
        trace("Friday");
        break;
    case 6:
        trace("Saturday");
        break;
    default:
        trace("Out of range");
        break;
}
```

Looping

Looping statements allow you to perform a specific block of code repeatedly using a series of values or variables. Adobe recommends that you always enclose the block of code in curly brackets ({}). Although you can omit the curly brackets if the block of code contains only one statement, this practice is not recommended for the same reason that it is not recommended for conditionals: it increases the likelihood that statements added later are inadvertently excluded from the block of code. If you later add a statement that you want to include in the block of code, but forget to add the necessary curly brackets, the statement are not executed as part of the loop.

for

The `for` loop allows you to iterate through a variable for a specific range of values. You must supply three expressions in a `for` statement: a variable that is set to an initial value, a conditional statement that determines when the looping ends, and an expression that changes the value of the variable with each loop. For example, the following code loops five times. The value of the variable `i` starts at 0 and ends at 4, and the output is the numbers 0 through 4, each on its own line.

```
var i:int;
for (i = 0; i < 5; i++)
{
    trace(i);
}
```

for..in

The `for..in` loop iterates through the properties of an object, or the elements of an array. For example, you can use a `for..in` loop to iterate through the properties of a generic object (object properties are not kept in any particular order, so properties may appear in a seemingly random order):

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
    trace(i + ": " + myObj[i]);
}
// output:
// x: 20
// y: 30
```

You can also iterate through the elements of an array:

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
{
    trace(myArray[i]);
}
// output:
// one
// two
// three
```

What you cannot do is iterate through the properties of an object if it is an instance of a sealed class (including built-in classes and user-defined classes). You can only iterate through the properties of a dynamic class. Even with instances of dynamic classes, you can only iterate through properties that are added dynamically.

for each..in

The `for each..in` loop iterates through the items of a collection, which can be tags in an XML or XMLList object, the values held by object properties, or the elements of an array. For example, as the following excerpt shows, you can use a `for each..in` loop to iterate through the properties of a generic object, but unlike the `for..in` loop, the iterator variable in a `for each..in` loop contains the value held by the property instead of the name of the property:

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
    trace(num);
}
// output:
// 20
// 30
```

You can iterate through an XML or XMLList object, as the following example shows:

```
var myXML:XML = <users>
    <fname>Jane</fname>
    <fname>Susan</fname>
    <fname>John</fname>
</users>;

for each (var item in myXML.fname)
{
    trace(item);
}
/* output
Jane
Susan
John
*/
```

You can also iterate through the elements of an array, as this example shows:

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
    trace(item);
}
// output:
// one
// two
// three
```

You cannot iterate through the properties of an object if the object is an instance of a sealed class. Even for instances of dynamic classes, you cannot iterate through any fixed properties, which are properties defined as part of the class definition.

while

The `while` loop is like an `if` statement that repeats as long as the condition is `true`. For example, the following code produces the same output as the `for` loop example:

```
var i:int = 0;
while (i < 5)
{
    trace(i);
    i++;
}
```

One disadvantage of using a `while` loop instead of a `for` loop is that infinite loops are easier to write with `while` loops. The `for` loop example code does not compile if you omit the expression that increments the counter variable, but the `while` loop example does compile if you omit that step. Without the expression that increments `i`, the loop becomes an infinite loop.

do..while

The `do..while` loop is a `while` loop that guarantees that the code block is executed at least once, because the condition is checked after the code block is executed. The following code shows a simple example of a `do..while` loop that generates output even though the condition is not met:

```
var i:int = 5;
do
{
    trace(i);
    i++;
} while (i < 5);
// output: 5
```

Functions

Functions are blocks of code that carry out specific tasks and can be reused in your program. There are two types of functions in ActionScript 3.0: *methods* and *function closures*. Whether a function is a called a method or a function closure depends on the context in which the function is defined. A function is called a method if you define it as part of a class definition or attach it to an instance of an object. A function is called a function closure if it is defined in any other way.

Functions have always been extremely important in ActionScript. In ActionScript 1.0, for example, the `class` keyword did not exist, so "classes" were defined by constructor functions. Although the `class` keyword has since been added to the language, a solid understanding of functions is still important if you want to take full advantage of what the language has to offer. This can be a challenge for programmers who expect ActionScript functions to behave similarly to functions in languages such as C++ or Java. Although basic function definition and invocation should not present a challenge to experienced programmers, some of the more advanced features of ActionScript functions require some explanation.

Basic function concepts

Calling functions

You call a function by using its identifier followed by the parentheses operator (`()`). You use the parentheses operator to enclose any function parameters you want to send to the function. For example, the `trace()` function is a top-level function in ActionScript 3.0:

```
trace("Use trace to help debug your script");
```

If you are calling a function with no parameters, you must use an empty pair of parentheses. For example, you can use the `Math.random()` method, which takes no parameters, to generate a random number:

```
var randomNum:Number = Math.random();
```

Defining your own functions

There are two ways to define a function in ActionScript 3.0: you can use a function statement or a function expression. The technique you choose depends on whether you prefer a more static or dynamic programming style. Define your functions with function statements if you prefer static, or strict mode, programming. Define your functions with function expressions if you have a specific need to do so. Function expressions are more often used in dynamic, or standard mode, programming.

Function statements

Function statements are the preferred technique for defining functions in strict mode. A function statement begins with the `function` keyword, followed by:

- The function name
- The parameters, in a comma-delimited list enclosed in parentheses
- The function body—that is, the ActionScript code to be executed when the function is called, enclosed in curly brackets

For example, the following code creates a function that defines a parameter and then calls the function using the string "`hello`" as the parameter value:

```
function traceParameter(aParam:String)
{
    trace(aParam);
}

traceParameter("hello"); // hello
```

Function expressions

The second way to declare a function is to use an assignment statement with a function expression, which is also sometimes called a function literal or an anonymous function. This is a more verbose method that is widely used in earlier versions of ActionScript.

An assignment statement with a function expression begins with the `var` keyword, followed by:

- The function name
- The colon operator (`:`)
- The `Function` class to indicate the data type
- The assignment operator (`=`)
- The `function` keyword
- The parameters, in a comma-delimited list enclosed in parentheses
- The function body—that is, the ActionScript code to be executed when the function is called, enclosed in curly brackets

  For example, the following code declares the `traceParameter` function using a function expression:

  ```
  var traceParameter:Function = function (aParam:String)
  {
      trace(aParam);
  };
  traceParameter("hello"); // hello
  ```

Notice that you do not specify a function name, as you do in a function statement. Another important difference between function expressions and function statements is that a function expression is an expression rather than a statement. This means that a function expression cannot stand on its own as a function statement can. A function expression can be used only as a part of a statement, usually an assignment statement. The following example shows a function expression assigned to an array element:

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
    trace(aParam);
};
traceArray[0]("hello");
```

Choosing between statements and expressions

As a general rule, use a function statement unless specific circumstances call for the use of an expression. Function statements are less verbose, and they provide a more consistent experience between strict mode and standard mode than function expressions.

Function statements are easier to read than assignment statements that contain function expressions. Function statements make your code more concise; they are less confusing than function expressions, which require you to use both the `var` and `function` keywords.

Function statements provide a more consistent experience between the two compiler modes in that you can use dot syntax in both strict and standard mode to call a method declared using a function statement. This is not necessarily true for methods declared with a function expression. For example, the following code defines a class named Example with two methods: `methodExpression()`, which is declared with a function expression, and `methodStatement()`, which is declared with a function statement. In strict mode, you cannot use dot syntax to call the `methodExpression()` method.

```
class Example
{
var methodExpression = function() {}
function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

Function expressions are considered better suited to programming that focuses on run-time, or dynamic, behavior. If you prefer to use strict mode, but also need to call a method declared with a function expression, you can use either of two techniques. First, you can call the method using square brackets (`[]`) instead of the dot (.) operator. The following method call succeeds in both strict mode and standard mode:

```
myExample["methodLiteral"]();
```

Second, you can declare the entire class as a dynamic class. Although this allows you to call the method using the dot operator, the downside is that you sacrifice some strict mode functionality for all instances of that class. For example, the compiler does not generate an error if you attempt to access an undefined property on an instance of a dynamic class.

There are some circumstances in which function expressions are useful. One common use of function expressions is for functions that are used only once and then discarded. Another less common use is for attaching a function to a prototype property. For more information, see The prototype object.

There are two subtle differences between function statements and function expressions that you should take into account when choosing which technique to use. The first difference is that function expressions do not exist independently as objects with regard to memory management and garbage collection. In other words, when you assign a function expression to another object, such as an array element or an object property, you create the only reference to that function expression in your code. If the array or object to which your function expression is attached goes out of scope or is otherwise no longer available, you no longer have access to the function expression. If the array or object is deleted, the memory that the function expression uses becomes eligible for garbage collection, which means that the memory is eligible to be reclaimed and reused for other purposes.

The following example shows that for a function expression, once the property to which the expression is assigned is deleted, the function is no longer available. The class Test is dynamic, which means that you can add a property named `functionExp` that holds a function expression. The `functionExp()` function can be called with the dot operator, but once the `functionExp` property is deleted, the function is no longer accessible.

```
dynamic class Test {}
var myTest:Test = new Test();

// function expression
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp(); // Function expression
delete myTest.functionExp;
myTest.functionExp(); // error
```

If, on the other hand, the function is first defined with a function statement, it exists as its own object and continues to exist even after you delete the property to which it is attached. The `delete` operator only works on properties of objects, so even a call to delete the function `stateFunc()` itself does not work.

```
dynamic class Test {}
var myTest:Test = new Test();

// function statement
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // Function statement
delete myTest.statement;
delete stateFunc; // no effect
stateFunc();// Function statement
myTest.statement(); // error
```

The second difference between function statements and function expressions is that function statements exist throughout the scope in which they are defined, including in statements that appear before the function statement. Function expressions, by contrast, are defined only for subsequent statements. For example, the following code successfully calls the `scopeTest()` function before it is defined:

```
statementTest(); // statementTest

function statementTest():void
{
    trace("statementTest");
}
```

Function expressions are not available before they are defined, so the following code results in a run-time error:

```
expressionTest(); // run-time error

var expressionTest:Function = function ()
{
    trace("expressionTest");
}
```

Returning values from functions

To return a value from your function, use the `return` statement followed by the expression or literal value that you want to return. For example, the following code returns an expression representing the parameter:

```
function doubleNum(baseNum:int):int
{
    return (baseNum * 2);
}
```

Notice that the `return` statement terminates the function, so that any statements below a `return` statement are not executed, as follows:

```
function doubleNum(baseNum:int):int {
    return (baseNum * 2);
    trace("after return"); // This trace statement will not be executed.
}
```

In strict mode, you must return a value of the appropriate type if you choose to specify a return type. For example, the following code generates an error in strict mode, because it does not return a valid value:

```
function doubleNum(baseNum:int):int
{
    trace("after return");
}
```

Nested functions

You can nest functions, which means that functions can be declared within other functions. A nested function is available only within its parent function unless a reference to the function is passed to external code. For example, the following code declares two nested functions inside the `getNameAndVersion()` function:

```
function getNameAndVersion():String
{
    function getVersion():String
    {
        return "10";
    }
    function getProductName():String
    {
        return "Flash Player";
    }
    return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 10
```

When nested functions are passed to external code, they are passed as function closures, which means that the function retains any definitions that are in scope when the function is defined. For more information, see Function scope.

Function parameters

ActionScript 3.0 provides some functionality for function parameters that may seem novel for programmers new to the language. Although the idea of passing parameters by value or reference should be familiar to most programmers, the `arguments` object and the ... (rest) parameter may be new to many of you.

Passing arguments by value or by reference

In many programming languages, it's important to understand the distinction between passing arguments by value or by reference; the distinction can affect the way code is designed.

To be passed by value means that the value of the argument is copied into a local variable for use within the function. To be passed by reference means that only a reference to the argument is passed instead of the actual value. No copy of the actual argument is made. Instead, a reference to the variable passed as an argument is created and assigned to a local variable for use within the function. As a reference to a variable outside the function, the local variable gives you the ability to change the value of the original variable.

In ActionScript 3.0, all arguments are passed by reference, because all values are stored as objects. However, objects that belong to the primitive data types, which includes Boolean, Number, int, uint, and String, have special operators that make them behave as if they were passed by value. For example, the following code creates a function named `passPrimitives()` that defines two parameters named `xParam` and `yParam`, both of type int. These parameters are similar to local variables declared inside the body of the `passPrimitives()` function. When the function is called with the arguments `xValue` and `yValue`, the parameters `xParam` and `yParam` are initialized with references to the int objects represented by `xValue` and `yValue`. Because the arguments are primitives, they behave as if passed by value. Although `xParam` and `yParam` initially contain only references to the `xValue` and `yValue` objects, any changes to the variables within the function body generate new copies of the values in memory.

```
function passPrimitives(xParam:int, yParam:int):void
{
    xParam++;
    yParam++;
    trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);// 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);// 10 15
```

Within the `passPrimitives()` function, the values of `xParam` and `yParam` are incremented, but this does not affect the values of `xValue` and `yValue`, as shown in the last `trace` statement. This would be true even if the parameters were named identically to the variables, `xValue` and `yValue`, because the `xValue` and `yValue` inside the function would point to new locations in memory that exist separately from the variables of the same name outside the function.

All other objects—that is, objects that do not belong to the primitive data types—are always passed by reference, which gives you ability to change the value of the original variable. For example, the following code creates an object named `objVar` with two properties, `x` and `y`. The object is passed as an argument to the `passByRef()` function. Because the object is not a primitive type, the object is not only passed by reference, but also stays a reference. This means that changes made to the parameters within the function affect the object properties outside the function.

```
function passByRef(objParam:Object):void
{
    objParam.x++;
    objParam.y++;
    trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar); // 11 16
trace(objVar.x, objVar.y); // 11 16
```

The `objParam` parameter references the same object as the global `objVar` variable. As you can see from the `trace` statements in the example, changes to the `x` and `y` properties of the `objParam` object are reflected in the `objVar` object.

Default parameter values

In ActionScript 3.0, you can declare *default parameter values* for a function. If a call to a function with default parameter values omits a parameter with default values, the value specified in the function definition for that parameter is used. All parameters with default values must be placed at the end of the parameter list. The values assigned as default values must be compile-time constants. The existence of a default value for a parameter effectively makes that parameter an *optional parameter*. A parameter without a default value is considered a *required parameter*.

For example, the following code creates a function with three parameters, two of which have default values. When the function is called with only one parameter, the default values for the parameters are used.

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
    trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

The arguments object

When parameters are passed to a function, you can use the `arguments` object to access information about the parameters passed to your function. Some important aspects of the `arguments` object include the following:

- The `arguments` object is an array that includes all the parameters passed to the function.
- The `arguments.length` property reports the number of parameters passed to the function.
- The `arguments.callee` property provides a reference to the function itself, which is useful for recursive calls to function expressions.

  Note: *The `arguments` object is not available if any parameter is named `arguments` or if you use the ... (rest) parameter.*

  If the `arguments` object is referenced in the body of a function, ActionScript 3.0 allows function calls to include more parameters than those defined in the function definition, but generates a compiler error in strict mode if the number of parameters doesn't match the number of required parameters (and optionally, any optional parameters). You can use the array aspect of the `arguments` object to access any parameter passed to the function, whether or not that parameter is defined in the function definition. The following example, which only compiles in standard mode, uses the `arguments` array along with the `arguments.length` property to trace all the parameters passed to the `traceArgArray()` function:

```
function traceArgArray(x:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

The `arguments.callee` property is often used in anonymous functions to create recursion. You can use it to add flexibility to your code. If the name of a recursive function changes over the course of your development cycle, you need not worry about changing the recursive call in your function body if you use `arguments.callee` instead of the function name. The `arguments.callee` property is used in the following function expression to enable recursion:

```
var factorial:Function = function (x:uint)
{
    if(x == 0)
    {
        return 1;
    }
    else
    {
        return (x * arguments.callee(x - 1));
    }
}

trace(factorial(5)); // 120
```

If you use the ... (rest) parameter in your function declaration, the `arguments` object is not available to you. Instead, you must access the parameters using the parameter names that you declared for them.

You should also be careful to avoid using the string `"arguments"` as a parameter name, because it shadows the `arguments` object. For example, if the function `traceArgArray()` is rewritten so that an `arguments` parameter is added, the references to `arguments` in the function body refer to the parameter rather than the `arguments` object. The following code produces no output:

```
function traceArgArray(x:int, arguments:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// no output
```

The `arguments` object in previous versions of ActionScript also contained a property named `caller`, which is a reference to the function that called the current function. The `caller` property is not present in ActionScript 3.0, but if you need a reference to the calling function, you can alter the calling function so that it passes an extra parameter that is a reference to itself.

The ... (rest) parameter

ActionScript 3.0 introduces a new parameter declaration called the ... (rest) parameter. This parameter allows you to specify an array parameter that accepts any number of comma- delimited arguments. The parameter can have any name that is not a reserved word. This parameter declaration must be the last parameter specified. Use of this parameter makes the `arguments` object unavailable. Although the ... (rest) parameter gives you the same functionality as the `arguments` array and `arguments.length` property, it does not provide functionality similar to that provided by `arguments.callee`. You should ensure that you do not need to use `arguments.callee` before using the ... (rest) parameter.

The following example rewrites the `traceArgArray()` function using the ... (rest) parameter instead of the `arguments` object:

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

The ... (rest) parameter can also be used with other parameters, as long as it is the last parameter listed. The following example modifies the `traceArgArray()` function so that its first parameter, `x`, is of type int, and the second parameter uses the ... (rest) parameter. The output skips the first value, because the first parameter is no longer part of the array created by the ... (rest) parameter.

```
function traceArgArray(x: int, ... args)
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

Functions as objects

Functions in ActionScript 3.0 are objects. When you create a function, you are creating an object that not only can be passed as a parameter to another function, but also can have properties and methods attached to it.

Functions passed as arguments to another function are passed by reference and not by value. When you pass a function as an argument, you use only the identifier and not the parentheses operator that you use to call the method. For example, the following code passes a function named `clickListener()` as an argument to the `addEventListener()` method:

```
addEventListener(MouseEvent.CLICK, clickListener);
```

Although it may seem strange to programmers new to ActionScript, functions can have properties and methods, just as any other object can. In fact, every function has a read-only property named `length` that stores the number of parameters defined for the function. This is different from the `arguments.length` property, which reports the number of arguments sent to the function. Recall that in ActionScript, the number of arguments sent to a function can exceed the number of parameters defined for that function. The following example, which compiles only in standard mode because strict mode requires an exact match between the number of arguments passed and the number of parameters defined, shows the difference between the two properties:

```
// Compiles only in standard mode
function traceLength(x:uint, y:uint):void
{
    trace("arguments received: " + arguments.length);
    trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

In standard mode, you can define your own function properties by defining them outside your function body. Function properties can serve as quasi-static properties that allow you to save the state of a variable related to the function. For example, you may want to track the number of times a particular function is called. Such functionality could be useful if you are writing a game and want to track the number of times a user uses a specific command, although you could also use a static class property for this. The following example, which compiles only in standard mode because strict mode does not allow you to add dynamic properties to functions, creates a function property outside the function declaration and increments the property each time the function is called:

```
// Compiles only in standard mode
var someFunction:Function = function ():void
{
    someFunction.counter++;
}

someFunction.counter = 0;

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

Function scope

A function's scope determines not only where in a program that function can be called, but also what definitions the function can access. The same scope rules that apply to variable identifiers apply to function identifiers. A function declared in the global scope is available throughout your code. For example, ActionScript 3.0 contains global functions, such as `isNaN()` and `parseInt()`, that are available anywhere in your code. A nested function—a function declared within another function—can be used anywhere in the function in which it was declared.

The scope chain

Any time a function begins execution, a number of objects and properties are created. First, a special object called an *activation object* is created that stores the parameters and any local variables or functions declared in the function body. You cannot access the activation object directly, because it is an internal mechanism. Second, a *scope chain* is created that contains an ordered list of objects that the runtime checks for identifier declarations. Every function that executes has a scope chain that is stored in an internal property. For a nested function, the scope chain starts with its own activation object, followed by its parent function's activation object. The chain continues in this manner until it reaches the global object. The global object is created when an ActionScript program begins, and contains all global variables and functions.

Function closures

A *function closure* is an object that contains a snapshot of a function and its *lexical environment*. A function's lexical environment includes all the variables, properties, methods, and objects in the function's scope chain, along with their values. Function closures are created any time a function is executed apart from an object or a class. The fact that function closures retain the scope in which they were defined 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 value `160` instead of `8`.

```
function foo():Function
{
    var x:int = 40;
    function rectArea(y:int):int // function closure defined
    {
        return x * y
    }
    return rectArea;
}
function bar():void
{
    var x:int = 2;
    var y:int = 4;
    var myProduct:Function = foo();
    trace(myProduct(4)); // function closure called
}
bar(); // 160
```

Methods behave similarly in that they also retain information about the lexical environment in which they were created. This characteristic is most noticeable when a method is extracted from its instance, which creates a bound method. The main difference between a function closure and a bound method is that the value of the `this` keyword in a bound method always refers to the instance to which it was originally attached, whereas in a function closure the value of the `this` keyword can change.

Chapter 4: Object-oriented programming in ActionScript

Introduction to object-oriented programming

Object-oriented programming (OOP) is a way of organizing the code in a program by grouping it into objects. The term *object* in this sense means an individual element that include information (data values) and functionality. When you use an object-oriented approach to organizing a program you group particular pieces of information together with common functionality or actions associated with that information. For example, you could group music information like album title, track title, or artist name with functionality such as "add track to playlist" or "play all songs by this artist". These pieces are combined into a single item, an object (for example, an "Album" or "MusicTrack"). Bundling values and functions together provides several benefits. One key benefit is that you only needing to use a single variable rather than multiple ones. In addition, it keeps related functionality together. Finally, combining information and functionality allows you to structure programs in ways that more closely match the real world.

Classes

A class is an abstract representation of an object. A class stores information about the types of data that an object can hold and the behaviors that an object can exhibit. The usefulness of such an abstraction is not necessarily apparent when you write small scripts that contain only a few objects interacting with one another. However, as the scope of a program grows the number of objects that must be managed increases. In that case classes allow you to better control how objects are created and how they interact with one another.

As far back as ActionScript 1.0, ActionScript programmers could use Function objects to create constructs that resembled classes. ActionScript 2.0 added formal support for classes with keywords such as `class` and `extends`. ActionScript 3.0 not only continues to support the keywords introduced in ActionScript 2.0. It also adds new capabilities. For example, ActionScript 3.0 includes enhanced access control with the `protected` and `internal` attributes. It also provides better control over inheritance with the `final` and `override` keywords.

For developers who have created classes in programming languages like Java, C++, or C#, ActionScript provides a familiar experience. ActionScript shares many of the same keywords and attribute names, such as `class`, `extends`, and `public`.

Note: *In the Adobe ActionScript documentation, the term property means any member of an object or class, including variables, constants, and methods. In addition, although the terms class and static are often used interchangeably, here these terms are distinct. For example, the phrase "class properties" is used to mean all the members of a class, rather than only the static members.*

Class definitions

ActionScript 3.0 class definitions use syntax that is similar to the syntax used in ActionScript 2.0 class definitions. Proper syntax for a class definition calls for the `class` keyword followed by the class name. The class body, enclosed by curly brackets ({}), follows the class name. For example, the following code creates a class named Shape that contains one variable, named `visible`:

```
public class Shape
{
    var visible:Boolean = true;
}
```

One significant syntax change involves class definitions that are inside a package. In ActionScript 2.0, if a class is inside a package, the package name must be included in the class declaration. In ActionScript 3.0, which introduces the `package` statement, the package name must be included in the package declaration instead of in the class declaration. For example, the following class declarations show how the BitmapData class, which is part of the flash.display package, is defined in ActionScript 2.0 and ActionScript 3.0:

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

Class attributes

ActionScript 3.0 allows you to modify class definitions using one of the following four attributes:

Attribute	Definition
`dynamic`	Allow properties to be added to instances at run time.
`final`	Must not be extended by another class.
`internal` (default)	Visible to references inside the current package.
`public`	Visible to references everywhere.

For each of these attributes, except for `internal`, you explicitly include the attribute to get the associated behavior. For example, if you do not include the `dynamic` attribute when defining a class, you can't add properties to a class instance at run time. You explicitly assign an attribute by placing it at the beginning of the class definition, as the following code demonstrates:

```
dynamic class Shape {}
```

Notice that the list does not include an attribute named `abstract`. Abstract classes are not supported in ActionScript 3.0. Notice also that the list does not include attributes named `private` and `protected`. These attributes have meaning only inside a class definition, and cannot be applied to classes themselves. If you do not want a class to be publicly visible outside a package, place the class inside a package and mark the class with the `internal` attribute. Alternatively, you can omit both the `internal` and `public` attributes, and the compiler automatically adds the `internal` attribute for you. You can also define a class to only be visible inside the source file in which it is defined. Place the class at the bottom of your source file, below the closing curly bracket of the package definition.

Class body

The class body is enclosed by curly brackets. It defines the variables, constants, and methods of your class. The following example shows the declaration for the Accessibility class in ActionScript 3.0:

```
public final class Accessibility
{
    public static function get active():Boolean;
    public static function updateProperties():void;
}
```

You can also define a namespace inside a class body. The following example shows how a namespace can be defined within a class body and used as an attribute of a method in that class:

```
public class SampleClass
{
    public namespace sampleNamespace;
    sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 allows you to include not only definitions in a class body, but also statements. Statements that are inside a class body, but outside a method definition, are executed exactly once. This execution happens when the class definition is first encountered and the associated class object is created. The following example includes a call to an external function, `hello()`, and a `trace` statement that outputs a confirmation message when the class is defined:

```
function hello():String
{
    trace("hola");
}
class SampleClass
{
    hello();
    trace("class created");
}
// output when class is created
hola
class created
```

In ActionScript 3.0 it is permissible to define a static property and an instance property with the same name in the same class body. For example, the following code declares a static variable named `message` and an instance variable of the same name:

```
class StaticTest
{
    static var message:String = "static variable";
    var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // output: static variable
trace(myST.message); // output: instance variable
```

Class property attributes

In discussions of the ActionScript object model, the term *property* means anything that can be a member of a class, including variables, constants, and methods. However, in the Adobe ActionScript 3.0 Reference for the Adobe Flash Platform the term is used more narrowly. In that context the term property includes only class members that are variables or are defined by a getter or setter method. In ActionScript 3.0, there is a set of attributes that can be used with any property of a class. The following table lists this set of attributes.

Attribute	**Definition**
`internal` (default)	Visible to references inside the same package.
`private`	Visible to references in the same class.
`protected`	Visible to references in the same class and derived classes.

Attribute	Definition
`public`	Visible to references everywhere.
`static`	Specifies that a property belongs to the class, as opposed to instances of the class.
`UserDefinedNamespace`	Custom namespace name defined by user.

Access control namespace attributes

ActionScript 3.0 provides four special attributes that control access to properties defined inside a class: `public`, `private`, `protected`, and `internal`.

The `public` attribute makes a property visible anywhere in your script. For example, to make a method available to code outside its package, you must declare the method with the `public` attribute. This is true for any property, whether it is declared using the `var`, `const`, or `function` keywords.

The `private` attribute makes a property visible only to callers within the property's defining class. This behavior differs from that of the `private` attribute in ActionScript 2.0, which allowed a subclass to access a private property in a superclass. Another significant change in behavior has to do with run-time access. In ActionScript 2.0, the `private` keyword prohibited access only at compile time and was easily circumvented at run time. In ActionScript 3.0, this is no longer true. Properties that are marked as `private` are unavailable at both compile time and run time.

For example, the following code creates a simple class named PrivateExample with one private variable, and then attempts to access the private variable from outside the class.

```
class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in ActionScript 3.0, this
is a run-time error.
```

In ActionScript 3.0, an attempt to access a private property using the dot operator (`myExample.privVar`) results in a compile-time error if you are using strict mode. Otherwise, the error is reported at run time, just as it is when you use the property access operator (`myExample["privVar"]`).

The following table summarizes the results of attempting to access a private property that belongs to a sealed (not dynamic) class:

	Strict mode	Standard mode
dot operator (`.`)	compile-time error	run-time error
bracket operator (`[]`)	run-time error	run-time error

In classes declared with the `dynamic` attribute, attempts to access a private variable do not result in a run-time error. Instead, the variable is not visible, so the value `undefined` is returned. A compile-time error occurs, however, if you use the dot operator in strict mode. The following example is the same as the previous example, except that the PrivateExample class is declared as a dynamic class:

```
dynamic class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // output: undefined
```

Dynamic classes generally return the value `undefined` instead of generating an error when code external to a class attempts to access a private property. The following table shows that an error is generated only when the dot operator is used to access a private property in strict mode:

	Strict mode	**Standard mode**
dot operator (`.`)	compile-time error	`undefined`
bracket operator (`[]`)	`undefined`	`undefined`

The `protected` attribute, which is new for ActionScript 3.0, makes a property visible to callers within its own class or in a subclass. In other words, a protected property is available within its own class or to classes that lie anywhere below it in the inheritance hierarchy. This is true whether the subclass is in the same package or in a different package.

For those familiar with ActionScript 2.0, this functionality is similar to the `private` attribute in ActionScript 2.0. The ActionScript 3.0 `protected` attribute is also similar to the `protected` attribute in Java. It differs in that the Java version also permits access to callers within the same package. The `protected` attribute is useful when you have a variable or method that your subclasses need but that you want to hide from code that is outside the inheritance chain.

The `internal` attribute, which is new for ActionScript 3.0, makes a property visible to callers within its own package. This is the default attribute for code inside a package, and it applies to any property that does not have any of the following attributes:

- `public`
- `private`
- `protected`
- a user-defined namespace

The `internal` attribute is similar to the default access control in Java, although in Java there is no explicit name for this level of access, and it can be achieved only through the omission of any other access modifier. The `internal` attribute is available in ActionScript 3.0 to give you the option of explicitly signifying your intent to make a property visible only to callers within its own package.

static attribute

The `static` attribute, which can be used with properties declared with the `var`, `const`, or `function` keywords, allows you to attach a property to the class rather than to instances of the class. Code external to the class must call static properties by using the class name instead of an instance name.

Static properties are not inherited by subclasses, but the properties are part of a subclass's scope chain. This means that within the body of a subclass, a static variable or method can be used without referencing the class in which it was defined.

User-defined namespace attributes

As an alternative to the predefined access control attributes, you can create a custom namespace for use as an attribute. Only one namespace attribute can be used per definition, and you cannot use a namespace attribute in combination with any of the access control attributes (`public`, `private`, `protected`, `internal`).

Variables

Variables can be declared with either the `var` or `const` keywords. Variables declared with the `var` keyword can have their values changed multiple times throughout the execution of a script. Variables declared with the `const` keyword are called *constants*, and can have values assigned to them only once. An attempt to assign a new value to an initialized constant results in an error.

Static variables

Static variables are declared using a combination of the `static` keyword and either the `var` or `const` statement. Static variables, which are attached to a class rather than an instance of a class, are useful for storing and sharing information that applies to an entire class of objects. For example, a static variable is appropriate if you want to keep a tally of the number of times a class is instantiated or if you want to store the maximum number of class instances that are allowed.

The following example creates a `totalCount` variable to track the number of class instantiations and a `MAX_NUM` constant to store the maximum number of instantiations. The `totalCount` and `MAX_NUM` variables are static, because they contain values that apply to the class as a whole rather than to a particular instance.

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

Code that is external to the StaticVars class and any of its subclasses can reference the `totalCount` and `MAX_NUM` properties only through the class itself. For example, the following code works:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

You cannot access static variables through an instance of the class, so the following code returns errors:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Variables that are declared with both the `static` and `const` keywords must be initialized at the same time as you declare the constant, as the StaticVars class does for `MAX_NUM`. You cannot assign a value to `MAX_NUM` inside the constructor or an instance method. The following code generates an error, because it is not a valid way to initialize a static constant:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Instance variables

Instance variables include properties declared with the `var` and `const` keywords, but without the `static` keyword. Instance variables, which are attached to class instances rather than to an entire class, are useful for storing values that are specific to an instance. For example, the Array class has an instance property named `length`, which stores the number of array elements that a particular instance of the Array class holds.

Instance variables, whether declared as `var` or `const`, cannot be overridden in a subclass. You can, however, achieve functionality that is similar to overriding variables by overriding getter and setter methods.

Methods

Methods are functions that are part of a class definition. Once an instance of the class is created, a method is bound to that instance. Unlike a function declared outside a class, a method cannot be used apart from the instance to which it is attached.

Methods are defined using the `function` keyword. As with any class property, you can apply any of the class property attributes to methods, including private, protected, public, internal, static, or a custom namespace. You can use a function statement such as the following:

```
public function sampleFunction():String {}
```

Or you can use a variable to which you assign a function expression, as follows:

```
public var sampleFunction:Function = function () {}
```

In most cases, use a function statement instead of a function expression for the following reasons:

- Function statements are more concise and easier to read.
- Function statements allow you to use the `override` and `final` keywords.
- Function statements create a stronger bond between the identifier (the name of the function) and the code within the method body. Because the value of a variable can be changed with an assignment statement, the connection between a variable and its function expression can be severed at any time. Although you can work around this issue by declaring the variable with `const` instead of `var`, such a technique is not considered a best practice. It makes the code hard to read and prevents the use of the `override` and `final` keywords.

One case in which you must use a function expression is when you choose to attach a function to the prototype object.

Constructor methods

Constructor methods, sometimes called *constructors*, are functions that share the same name as the class in which they are defined. Any code that you include in a constructor method is executed whenever an instance of the class is created with the `new` keyword. For example, the following code defines a simple class named Example that contains a single property named `status`. The initial value of the `status` variable is set inside the constructor function.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Constructor methods can only be public, but the use of the `public` attribute is optional. You cannot use any of the other access control specifiers, including `private`, `protected`, or `internal`, on a constructor. You also cannot use a user-defined namespace with a constructor method.

A constructor can make an explicit call to the constructor of its direct superclass by using the `super()` statement. If the superclass constructor is not explicitly called, the compiler automatically inserts a call before the first statement in the constructor body. You can also call methods of the superclass by using the `super` prefix as a reference to the superclass. If you decide to use both `super()` and `super` in the same constructor body, be sure to call `super()` first. Otherwise, the `super` reference does not behave as expected. The `super()` constructor should also be called before any `throw` or `return` statement.

The following example demonstrates what happens if you attempt to use the `super` reference before calling the `super()` constructor. A new class, ExampleEx, extends the Example class. The ExampleEx constructor attempts to access the status variable defined in its superclass, but does so before calling `super()`. The `trace()` statement inside the ExampleEx constructor produces the value `null`, because the `status` variable is not available until the `super()` constructor executes.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Although it is legal to use the `return` statement inside a constructor, it is not permissible to return a value. In other words, `return` statements must not have associated expressions or values. Accordingly, constructor methods are not allowed to return values, which means that no return type can be specified.

If you do not define a constructor method in your class, the compiler automatically creates an empty constructor for you. If your class extends another class, the compiler includes a `super()` call in the constructor it generates.

Static methods

Static methods, also called *class methods*, are methods that are declared with the `static` keyword. Static methods, which are attached to a class rather than to an instance of a class, are useful for encapsulating functionality that affects something other than the state of an individual instance. Because static methods are attached to a class as a whole, static methods can be accessed only through a class and not through an instance of the class.

Static methods are useful for encapsulating functionality that is not limited to affecting the state of class instances. In other words, a method should be static if it provides functionality that does not directly affect the value of a class instance. For example, the Date class has a static method named `parse()`, which takes a string and converts it to a number. The method is static because it does not affect an individual instance of the class. Instead, the `parse()` method takes a string that represents a date value, parses the string, and returns a number in a format compatible with the internal representation of a Date object. This method is not an instance method, because it does not make sense to apply the method to an instance of the Date class.

Contrast the static `parse()` method with one of the instance methods of the Date class, such as `getMonth()`. The `getMonth()` method is an instance method, because it operates directly on the value of an instance by retrieving a specific component, the month, of a Date instance.

Because static methods are not bound to individual instances, you cannot use the keywords `this` or `super` within the body of a static method. Both the `this` reference and the `super` reference have meaning only within the context of an instance method.

In contrast with some other class-based programming languages, static methods in ActionScript 3.0 are not inherited.

Instance methods

Instance methods are methods that are declared without the `static` keyword. Instance methods, which are attached to instances of a class instead of the class as a whole, are useful for implementing functionality that affects individual instances of a class. For example, the Array class contains an instance method named `sort()`, which operates directly on Array instances.

Within the body of an instance method, both static and instance variables are in scope, which means that variables defined in the same class can be referenced using a simple identifier. For example, the following class, CustomArray, extends the Array class. The CustomArray class defines a static variable named `arrayCountTotal` to track the total number of class instances, an instance variable named `arrayNumber` that tracks the order in which the instances were created, and an instance method named `getPosition()` that returns the values of these variables.

```
public class CustomArray extends Array
{
    public static var arrayCountTotal:int = 0;
    public var arrayNumber:int;

    public function CustomArray()
    {
        arrayNumber = ++arrayCountTotal;
    }

    public function getArrayPosition():String
    {
         return ("Array " + arrayNumber + " of " + arrayCountTotal);
    }
}
```

Although code external to the class must access the `arrayCountTotal` static variable through the class object using `CustomArray.arrayCountTotal`, code that resides inside the body of the `getPosition()` method can refer directly to the static `arrayCountTotal` variable. This is true even for static variables in superclasses. Though static properties are not inherited in ActionScript 3.0, static properties in superclasses are in scope. For example, the Array class has a few static variables, one of which is a constant named `DESCENDING`. Code that resides in an Array subclass can access the static constant `DESCENDING` using a simple identifier:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

The value of the `this` reference within the body of an instance method is a reference to the instance to which the method is attached. The following code demonstrates that the `this` reference points to the instance that contains the method:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Inheritance of instance methods can be controlled with the keywords `override` and `final`. You can use the `override` attribute to redefine an inherited method, and the `final` attribute to prevent subclasses from overriding a method.

Get and set accessor methods

Get and set accessor functions, also called *getters* and *setters*, allow you to adhere to the programming principles of information hiding and encapsulation while providing an easy-to-use programming interface for the classes that you create. Get and set functions allow you to keep your class properties private to the class, but allow users of your class to access those properties as if they were accessing a class variable instead of calling a class method.

The advantage of this approach is that it allows you to avoid the traditional accessor functions with unwieldy names, such as `getPropertyName()` and `setPropertyName()`. Another advantage of getters and setters is that you can avoid having two public-facing functions for each property that allows both read and write access.

The following example class, named GetSet, includes get and set accessor functions named `publicAccess()` that provide access to the private variable named `privateProperty`:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

If you attempt to access the property `privateProperty` directly, an error occurs, as follows:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Instead, a user of the GetSet class uses something that appears to be a property named `publicAccess`, but that is really a pair of get and set accessor functions that operate on the private property named `privateProperty`. The following example instantiates the GetSet class, and then sets the value of the `privateProperty` using the public accessor named `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Getter and setter functions also make it possible to override properties that are inherited from a superclass, something that is not possible when you use regular class member variables. Class member variables that are declared using the `var` keyword cannot be overridden in a subclass. Properties that are created using getter and setter functions, however, do not have this restriction. You can use the `override` attribute on getter and setter functions that are inherited from a superclass.

Bound methods

A bound method, sometimes called a *method closure*, is simply a method that is extracted from its instance. Examples of bound methods include methods that are passed as arguments to a function or returned as values from a function. New in ActionScript 3.0, a bound method is similar to a function closure in that it retains its lexical environment even when extracted from its instance. The key difference, however, between a bound method and a function closure is that the `this` reference for a bound method remains linked, or bound, to the instance that implements the method. In other words, the `this` reference in a bound method always points to the original object that implemented the method. For function closures, the `this` reference is generic, which means that it points to whatever object the function is associated with at the time it is called.

Understanding bound methods is important if you use the `this` keyword. Recall that the `this` keyword provides a reference to a method's parent object. Most ActionScript programmers expect that the `this` keyword always represents the object or class that contains the definition of a method. Without method binding, however, this would not always be true. In previous versions of ActionScript, for example, the `this` reference did not always refer to the instance that implemented the method. When methods are extracted from an instance in ActionScript 2.0, not only is the `this` reference not bound to the original instance, but also the member variables and methods of the instance's class are not available. This is not a problem in ActionScript 3.0, because bound methods are automatically created when you pass a method as a parameter. Bound methods ensure that the `this` keyword always references the object or class in which a method is defined.

The following code defines a class named ThisTest, which contains a method named `foo()` that defines the bound method, and a method named `bar()` that returns the bound method. Code external to the class creates an instance of the ThisTest class, calls the `bar()` method, and stores the return value in a variable named `myFunc`.

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

The last two lines of code show that the `this` reference in the bound method `foo()` still points to an instance of ThisTest class, even though the `this` reference in the line just before it points to the global object. Moreover, the bound method stored in the `myFunc` variable still has access to the member variables of the ThisTest class. If this same code is run in ActionScript 2.0, the `this` references would match, and the `num` variable would be `undefined`.

One area where the addition of bound methods is most noticeable is with event handlers, because the `addEventListener()` method requires that you pass a function or method as an argument.

Enumerations with classes

Enumerations are custom data types that you create to encapsulate a small set of values. ActionScript 3.0 does not support a specific enumeration facility, unlike C++ with its `enum` keyword or Java with its Enumeration interface. You can, however, create enumerations using classes and static constants. For example, the PrintJob class in ActionScript 3.0 uses an enumeration named PrintJobOrientation to store the values `"landscape"` and `"portrait"`, as shown in the following code:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

By convention, an enumeration class is declared with the `final` attribute, because there is no need to extend the class. The class includes only static members, which means that you do not create instances of the class. Instead, you access the enumeration values directly through the class object, as shown in the following code excerpt:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

All of the enumeration classes in ActionScript 3.0 contain only variables of type String, int, or uint. The advantage of using enumerations instead of literal string or number values is that typographical mistakes are easier to find with enumerations. If you mistype the name of an enumeration, the ActionScript compiler generates an error. If you use literal values, the compiler does not complain if you spell a word incorrectly or use the wrong number. In the previous example, the compiler generates an error if the name of the enumeration constant is incorrect, as the following excerpt shows:

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

However, the compiler does not generate an error if you misspell a string literal value, as follows:

```
    if (pj.orientation == "portrai") // no compiler error
```

A second technique for creating enumerations also involves creating a separate class with static properties for the enumeration. This technique differs, however, in that each of the static properties contains an instance of the class instead of a string or integer value. For example, the following code creates an enumeration class for the days of the week:

```
public final class Day
{
    public static const MONDAY:Day = new Day();
    public static const TUESDAY:Day = new Day();
    public static const WEDNESDAY:Day = new Day();
    public static const THURSDAY:Day = new Day();
    public static const FRIDAY:Day = new Day();
    public static const SATURDAY:Day = new Day();
    public static const SUNDAY:Day = new Day();
}
```

This technique is not used by ActionScript 3.0 but is used by many developers who prefer the improved type checking that the technique provides. For example, a method that returns an enumeration value can restrict the return value to the enumeration data type. The following code shows not only a function that returns a day of the week, but also a function call that uses the enumeration type as a type annotation:

```
function getDay():Day
{
    var date:Date = new Date();
    var retDay:Day;
    switch (date.day)
    {
        case 0:
            retDay = Day.MONDAY;
            break;
        case 1:
            retDay = Day.TUESDAY;
            break;
        case 2:
            retDay = Day.WEDNESDAY;
            break;
        case 3:
            retDay = Day.THURSDAY;
            break;
        case 4:
            retDay = Day.FRIDAY;
            break;
        case 5:
            retDay = Day.SATURDAY;
            break;
        case 6:
            retDay = Day.SUNDAY;
            break;
    }
    return retDay;
}

var dayOfWeek:Day = getDay();
```

You can also enhance the Day class so that it associates an integer with each day of the week, and provides a `toString()` method that returns a string representation of the day.

Embedded asset classes

ActionScript 3.0 uses special classes, called *embedded asset classes*, to represent embedded assets. An *embedded asset* is an asset, such as a sound, image, or font, that is included in a SWF file at compile time. Embedding an asset instead of loading it dynamically ensures that it is available at run time, but at the cost of increased SWF file size.

Using embedded asset classes in Flash Professional

To embed an asset, first place the asset into a FLA file's library. Next, use the asset's linkage property to provide a name for the asset's embedded asset class. If a class by that name cannot be found in the classpath, a class is automatically generated for you. You can then create an instance of the embedded asset class and use any properties and methods defined or inherited by that class. For example, the following code can be used to play an embedded sound that is linked to an embedded asset class named PianoMusic:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

Alternatively, you can use the `[Embed]` metadata tag to embed assets in a Flash Professional project, described next. If you use the `[Embed]` metadata tag in your code, Flash Professional uses the Flex compiler to compile your project instead of the Flash Professional compiler.

Using embedded asset classes using the Flex compiler

If you are compiling your code using the Flex compiler, to embed an asset in ActionScript code use the `[Embed]` metadata tag. Place the asset in the main source folder or another folder that is in your project's build path. When the Flex compiler encounters an Embed metadata tag, it creates the embedded asset class for you. You can access the class through a variable of data type Class that you declare immediately following the `[Embed]` metadata tag. For example, the following code embeds a sound named sound1.mp3 and uses a variable named `soundCls` to store a reference to the embedded asset class associated with that sound. The example then creates an instance of the embedded asset class and calls the `play()` method on that instance:

```
package
{
    import flash.display.Sprite;
    import flash.media.SoundChannel;
    import mx.core.SoundAsset;

    public class SoundAssetExample extends Sprite
    {
        [Embed(source="sound1.mp3")]
        public var soundCls:Class;

        public function SoundAssetExample()
        {
            var mySound:SoundAsset = new soundCls() as SoundAsset;
            var sndChannel:SoundChannel = mySound.play();
        }
    }
}
```

Adobe Flash Builder

To use the `[Embed]` metadata tag in a Flash Builder ActionScript project, import any necessary classes from the Flex framework. For example, to embed sounds, import the mx.core.SoundAsset class. To use the Flex framework, include the file framework.swc in your ActionScript build path. This increases the size of your SWF file.

Adobe Flex
Alternatively, in Flex you can embed an asset with the `@Embed()` directive in an MXML tag definition.

Interfaces

An interface is a collection of method declarations that allows unrelated objects to communicate with one another. For example, ActionScript 3.0 defines the IEventDispatcher interface, which contains method declarations that a class can use to handle event objects. The IEventDispatcher interface establishes a standard way for objects to pass event objects to one another. The following code shows the definition of the IEventDispatcher interface:

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
            useCapture:Boolean=false, priority:int=0,
            useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
            useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

Interfaces are based on the distinction between a method's interface and its implementation. A method's interface includes all the information necessary to call that method, including the name of the method, all of its parameters, and its return type. A method's implementation includes not only the interface information, but also the executable statements that carry out the method's behavior. An interface definition contains only method interfaces, and any class that implements the interface is responsible for defining the method implementations.

In ActionScript 3.0, the EventDispatcher class implements the IEventDispatcher interface by defining all of the IEventDispatcher interface methods and adding method bodies to each of the methods. The following code is an excerpt from the EventDispatcher class definition:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

The IEventDispatcher interface serves as a protocol that EventDispatcher instances use to process event objects and pass them to other objects that have also implemented the IEventDispatcher interface.

Another way to describe an interface is to say that it defines a data type just as a class does. Accordingly, an interface can be used as a type annotation, just as a class can. As a data type, an interface can also be used with operators, such as the `is` and `as` operators, that require a data type. Unlike a class, however, an interface cannot be instantiated. This distinction has led many programmers to think of interfaces as abstract data types and classes as concrete data types.

Defining an interface

The structure of an interface definition is similar to that of a class definition, except that an interface can contain only methods with no method bodies. Interfaces cannot include variables or constants but can include getters and setters. To define an interface, use the `interface` keyword. For example, the following interface, IExternalizable, is part of the flash.utils package in ActionScript 3.0. The IExternalizable interface defines a protocol for serializing an object, which means converting an object into a format suitable for storage on a device or for transport across a network.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

The IExternalizable interface is declared with the `public` access control modifier. Interface definitions can only be modified by the `public` and `internal` access control specifiers. The method declarations inside an interface definition cannot have any access control specifiers.

ActionScript 3.0 follows a convention in which interface names begin with an uppercase `I`, but you can use any legal identifier as an interface name. Interface definitions are often placed at the top level of a package. Interface definitions cannot be placed inside a class definition or inside another interface definition.

Interfaces can extend one or more other interfaces. For example, the following interface, IExample, extends the IExternalizable interface:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Any class that implements the IExample interface must include implementations not only for the `extra()` method, but also for the `writeExternal()` and `readExternal()` methods inherited from the IExternalizable interface.

Implementing an interface in a class

A class is the only ActionScript 3.0 language element that can implement an interface. Use the `implements` keyword in a class declaration to implement one or more interfaces. The following example defines two interfaces, IAlpha and IBeta, and a class, Alpha, that implements them both:

```
interface IAlpha
{
    function foo(str:String):String;
}

interface IBeta
{
    function bar():void;
}

class Alpha implements IAlpha, IBeta
{
    public function foo(param:String):String {}
    public function bar():void {}
}
```

In a class that implements an interface, implemented methods must do the following:

- Use the `public` access control identifier.

- Use the same name as the interface method.
- Have the same number of parameters, each with data types that match the interface method parameter data types.
- Use the same return type.

```
public function foo(param:String):String {}
```

You do have some flexibility, however, in how you name the parameters of methods that you implement. Although the number of parameters and the data type of each parameter in the implemented method must match that of the interface method, the parameter names do not need to match. For example, in the previous example the parameter of the `Alpha.foo()` method is named `param`:

But the parameter is named `str` in the `IAlpha.foo()` interface method:

```
function foo(str:String):String;
```

You also have some flexibility with default parameter values. An interface definition can include function declarations with default parameter values. A method that implements such a function declaration must have a default parameter value that is a member of the same data type as the value specified in the interface definition, but the actual value does not have to match. For example, the following code defines an interface that contains a method with a default parameter value of 3:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

The following class definition implements the IGamma interface but uses a different default parameter value:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

The reason for this flexibility is that the rules for implementing an interface are designed specifically to ensure data type compatibility, and requiring identical parameter names and default parameter values is not necessary to achieve that objective.

Inheritance

Inheritance is a form of code reuse that allows programmers to develop new classes that are based on existing classes. The existing classes are often called *base classes* or *superclasses*, while the new classes are called *subclasses*. A key advantage of inheritance is that it allows you to reuse code from a base class yet leave the existing code unmodified. Moreover, inheritance requires no changes to the way that other classes interact with the base class. Rather than modifying an existing class that may have been thoroughly tested or may already be in use, using inheritance you can treat that class as an integrated module that you can extend with additional properties or methods. Accordingly, you use the `extends` keyword to indicate that a class inherits from another class.

Inheritance also allows you to take advantage of *polymorphism* in your code. Polymorphism is the ability to use a single method name for a method that behaves differently when applied to different data types. A simple example is a base class named Shape with two subclasses named Circle and Square. The Shape class defines a method named `area()`, which returns the area of the shape. If polymorphism is implemented, you can call the `area()` method on objects of type Circle and Square and have the correct calculations done for you. Inheritance enables polymorphism by allowing subclasses to inherit and redefine, or *override*, methods from the base class. In the following example, the `area()` method is redefined by the Circle and Square classes:

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

Because each class defines a data type, the use of inheritance creates a special relationship between a base class and a class that extends it. A subclass is guaranteed to possess all the properties of its base class, which means that an instance of a subclass can always be substituted for an instance of the base class. For example, if a method defines a parameter of type Shape, it is legal to pass an argument of type Circle because Circle extends Shape, as in the following:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Instance properties and inheritance

An instance property, whether defined with the `function`, `var`, or `const` keywords, is inherited by all subclasses as long as the property is not declared with the `private` attribute in the base class. For example, the Event class in ActionScript 3.0 has a number of subclasses that inherit properties common to all event objects.

For some types of events, the Event class contains all the properties necessary to define the event. These types of events do not require instance properties beyond those defined in the Event class. Examples of such events are the `complete` event, which occurs when data has loaded successfully, and the `connect` event, which occurs when a network connection has been established.

The following example is an excerpt from the Event class that shows some of the properties and methods that are inherited by subclasses. Because the properties are inherited, an instance of any subclass can access these properties.

```
public class Event
{
    public function get type():String;
    public function get bubbles():Boolean;
    ...

    public function stopPropagation():void {}
    public function stopImmediatePropagation():void {}
    public function preventDefault():void {}
    public function isDefaultPrevented():Boolean {}
    ...
}
```

Other types of events require unique properties not available in the Event class. These events are defined using subclasses of the Event class so that new properties can be added to the properties defined in the Event class. An example of such a subclass is the MouseEvent class, which adds properties unique to events associated with mouse movement or mouse clicks, such as the `mouseMove` and `click` events. The following example is an excerpt from the MouseEvent class that shows the definition of properties that exist on the subclass but not on the base class:

```
public class MouseEvent extends Event
{
    public static const CLICK:String= "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

Access control specifiers and inheritance

If a property is declared with the `public` keyword, the property is visible to code anywhere. This means that the `public` keyword, unlike the `private`, `protected`, and `internal` keywords, places no restrictions on property inheritance.

If a property is declared with `private` keyword, it is visible only in the class that defines it, which means that it is not inherited by any subclasses. This behavior is different from previous versions of ActionScript, where the `private` keyword behaved more like the ActionScript 3.0 `protected` keyword.

The `protected` keyword indicates that a property is visible not only within the class that defines it, but also to all subclasses. Unlike the `protected` keyword in the Java programming language, the `protected` keyword in ActionScript 3.0 does not make a property visible to all other classes in the same package. In ActionScript 3.0, only subclasses can access a property declared with the `protected` keyword. Moreover, a protected property is visible to a subclass whether the subclass is in the same package as the base class or in a different package.

To limit the visibility of a property to the package in which it is defined, use the `internal` keyword or do not use any access control specifier. The `internal` access control specifier is the default access control specifier that applies when one is not specified. A property marked as `internal` is only inherited by a subclass that resides in the same package.

You can use the following example to see how each of the access control specifiers affects inheritance across package boundaries. The following code defines a main application class named AccessControl and two other classes named Base and Extender. The Base class is in a package named foo and the Extender class, which is a subclass of the Base class, is in a package named bar. The AccessControl class imports only the Extender class and creates an instance of Extender that attempts to access a variable named `str` that is defined in the Base class. The `str` variable is declared as `public` so that the code compiles and runs as shown in the following excerpt:

```
// Base.as in a folder named foo
package foo
{
    public class Base
    {
        public var str:String = "hello"; // change public on this line
    }
}

// Extender.as in a folder named bar
package bar
{
    import foo.Base;
    public class Extender extends Base
    {
        public function getString():String {
            return str;
        }
    }
}

// main application class in file named AccessControl.as
package
{
    import flash.display.MovieClip;
    import bar.Extender;
    public class AccessControl extends MovieClip
    {
        public function AccessControl()
        {
            var myExt:Extender = new Extender();
            trace(myExt.str);// error if str is not public
            trace(myExt.getString()); // error if str is private or internal
        }
    }
}
```

To see how the other access control specifiers affect compilation and execution of the preceding example, change the `str` variable's access control specifier to `private`, `protected`, or `internal` after deleting or commenting out the following line from the `AccessControl` class:

```
trace(myExt.str);// error if str is not public
```

Overriding variables not permitted

Properties that are declared with the `var` or `const` keywords are inherited but cannot be overridden. To override a property means to redefine the property in a subclass. The only type of property that can be overridden are get and set accessors (properties declared with the `function` keyword). Although you cannot override an instance variable, you can achieve similar functionality by creating getter and setter methods for the instance variable and overriding the methods.

Overriding methods

To override a method means to redefine the behavior of an inherited method. Static methods are not inherited and cannot be overridden. Instance methods, however, are inherited by subclasses and can be overridden as long as the following two criteria are met:

- The instance method is not declared with the `final` keyword in the base class. When used with an instance method, the `final` keyword indicates the programmer's intent to prevent subclasses from overriding the method.
- The instance method is not declared with the `private` access control specifier in the base class. If a method is marked as `private` in the base class, there is no need to use the `override` keyword when defining an identically named method in the subclass, because the base class method is not visible to the subclass.

  To override an instance method that meets these criteria, the method definition in the subclass must use the `override` keyword and must match the superclass version of the method in the following ways:

- The override method must have the same level of access control as the base class method. Methods marked as internal have the same level of access control as methods that have no access control specifier.
- The override method must have the same number of parameters as the base class method.
- The override method parameters must have the same data type annotations as the parameters in the base class method.
- The override method must have the same return type as the base class method.

The names of the parameters in the override method, however, do not have to match the names of the parameters in the base class, as long as the number of parameters and the data type of each parameter matches.

The super statement

When overriding a method, programmers often want to add to the behavior of the superclass method they are overriding instead of completely replacing the behavior. This requires a mechanism that allows a method in a subclass to call the superclass version of itself. The `super` statement provides such a mechanism, in that it contains a reference to the immediate superclass. The following example defines a class named Base that contains a method named `thanks()` and a subclass of the Base class named Extender that overrides the `thanks()` method. The `Extender.thanks()` method uses the `super` statement to call `Base.thanks()`.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

Overriding getters and setters

Although you cannot override variables defined in a superclass, you can override getters and setters. For example, the following code overrides a getter named `currentLabel` that is defined in the MovieClip class in ActionScript 3.0.:

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

The output of the `trace()` statement in the OverrideExample class constructor is `Override: null`, which shows that the example was able to override the inherited `currentLabel` property.

Static properties not inherited

Static properties are not inherited by subclasses. This means that static properties cannot be accessed through an instance of a subclass. A static property can be accessed only through the class object on which it is defined. For example, the following code defines a base class named Base and a subclass that extends Base named Extender. A static variable named `test` is defined in the Base class. The code as written in the following excerpt does not compile in strict mode and generates a run-time error in standard mode.

```
package {
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// error
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base { }
```

The only way to access the static variable `test` is through the class object, as shown in the following code:

```
Base.test;
```

It is permissible, however, to define an instance property using the same name as a static property. Such an instance property can be defined in the same class as the static property or in a subclass. For example, the Base class in the preceding example could have an instance property named `test`. The following code compiles and executes because the instance property is inherited by the Extender class. The code would also compile and execute if the definition of the test instance variable is moved, but not copied, to the Extender class.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

Static properties and the scope chain

Although static properties are not inherited, they are within the scope chain of the class that defines them and any subclass of that class. As such, static properties are said to be *in scope* of both the class in which they are defined and any subclasses. This means that a static property is directly accessible within the body of the class that defines the static property and any subclass of that class.

The following example modifies the classes defined in the previous example to show that the static `test` variable defined in the Base class is in scope of the Extender class. In other words, the Extender class can access the static `test` variable without prefixing the variable with the name of the class that defines `test`.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base
{
    public function Extender()
    {
        trace(test); // output: static
    }

}
```

If an instance property is defined that uses the same name as a static property in the same class or a superclass, the instance property has higher precedence in the scope chain. The instance property is said to *shadow* the static property, which means that the value of the instance property is used instead of the value of the static property. For example, the following code shows that if the Extender class defines an instance variable named `test`, the `trace()` statement uses the value of the instance variable instead of the value of the static variable.:

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base
{
    public static var test:String = "static";
}

class Extender extends Base
{
    public var test:String = "instance";
    public function Extender()
    {
        trace(test); // output: instance
    }

}
```

Advanced topics

History of ActionScript OOP support

Because ActionScript 3.0 builds upon previous versions of ActionScript, it may be helpful to understand how the ActionScript object model has evolved. ActionScript began as a simple scripting mechanism for early versions of Flash Professional. Later, programmers began building increasingly complex applications with ActionScript. In response to the needs of such programmers, each subsequent release has added language features that facilitate the creation of complex applications.

ActionScript 1.0

ActionScript 1.0 is the version of the language used in Flash Player 6 and earlier. Even at this early stage of development, the ActionScript object model was based on the concept of the object as a fundamental data type. An ActionScript object is a compound data type with a group of *properties*. When discussing the object model, the term *properties* includes everything that is attached to an object, such as variables, functions, or methods.

Although this first generation of ActionScript does not support the definition of classes with a `class` keyword, you can define a class using a special kind of object called a prototype object. Instead of using a `class` keyword to create an abstract class definition that you instantiate into concrete objects, as you do in class-based languages like Java and C++, prototype-based languages like ActionScript 1.0 use an existing object as a model (or prototype) for other objects. While objects in a class-based language may point to a class that serves as its template, objects in a prototype-based language point instead to another object, its prototype, that serves as its template.

To create a class in ActionScript 1.0, you define a constructor function for that class. In ActionScript, functions are actual objects, not just abstract definitions. The constructor function that you create serves as the prototypical object for instances of that class. The following code creates a class named Shape and defines one property named `visible` that is set to `true` by default:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

This constructor function defines a Shape class that you can instantiate with the `new` operator, as follows:

```
myShape = new Shape();
```

Just as the `Shape()` constructor function object serves as the prototype for instances of the Shape class, it can also serve as the prototype for subclasses of Shape—that is, other classes that extend the Shape class.

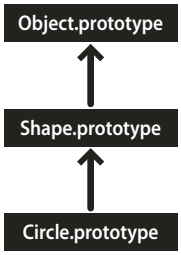
The creation of a class that is a subclass of the Shape class is a two-step process. First, create the class by defining a constructor function for the class, as follows:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

Second, use the `new` operator to declare that the Shape class is the prototype for the Circle class. By default, any class you create uses the Object class as its prototype, which means that `Circle.prototype` currently contains a generic object (an instance of the Object class). To specify that Circle's prototype is Shape instead of Object, use the following code to change the value of `Circle.prototype` so that it contains a Shape object instead of a generic object:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

The Shape class and the Circle class are now linked together in an inheritance relationship that is commonly known as the *prototype chain*. The diagram illustrates the relationships in a prototype chain:



The base class at the end of every prototype chain is the Object class. The Object class contains a static property named `Object.prototype` that points to the base prototype object for all objects created in ActionScript 1.0. The next object in the example prototype chain is the Shape object. This is because the `Shape.prototype` property was never explicitly set, so it still holds a generic object (an instance of the Object class). The final link in this chain is the Circle class, which is linked to its prototype, the Shape class (the `Circle.prototype` property holds a Shape object).

If you create an instance of the Circle class, as in the following example, the instance inherits the prototype chain of the Circle class:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Recall that the example included a property named `visible` as a member of the Shape class. In this example, the `visible` property does not exist as a part of the `myCircle` object, only as a member of the Shape object, yet the following line of code outputs `true`:

```
trace(myCircle.visible); // output: true
```

The runtime is able to ascertain that the `myCircle` object inherits the `visible` property by walking up the prototype chain. When executing this code, the runtime first searches through the properties of the `myCircle` object for a property named `visible`, but does not find such a property. It looks next in the `Circle.prototype` object, but still does not find a property named `visible`. Continuing up the prototype chain, it finally finds the `visible` property defined on the `Shape.prototype` object and outputs the value of that property.

In the interest of simplicity, many of the details and intricacies of the prototype chain are omitted. Instead the goal is to provide enough information to help you understand the ActionScript 3.0 object model.

ActionScript 2.0

ActionScript 2.0 introduced new keywords such as `class`, `extends`, `public`, and `private`, that allowed you to define classes in a way that is familiar to anyone who works with class-based languages like Java and C++. It's important to understand that the underlying inheritance mechanism did not change between ActionScript 1.0 and ActionScript 2.0. ActionScript 2.0 merely added a new syntax for defining classes. The prototype chain works the same way in both versions of the language.

The new syntax introduced by ActionScript 2.0, shown in the following excerpt, allows you to define classes in a way that many programmers find more intuitive:

```
// base class
class Shape
{
var visible:Boolean = true;
}
```

Note that ActionScript 2.0 also introduced type annotations for use with compile-time type checking. This allows you to declare that the `visible` property in the previous example should contain only a Boolean value. The new `extends` keyword also simplifies the process of creating a subclass. In the following example, the two-step process necessary in ActionScript 1.0 is accomplished in one step with the `extends` keyword:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

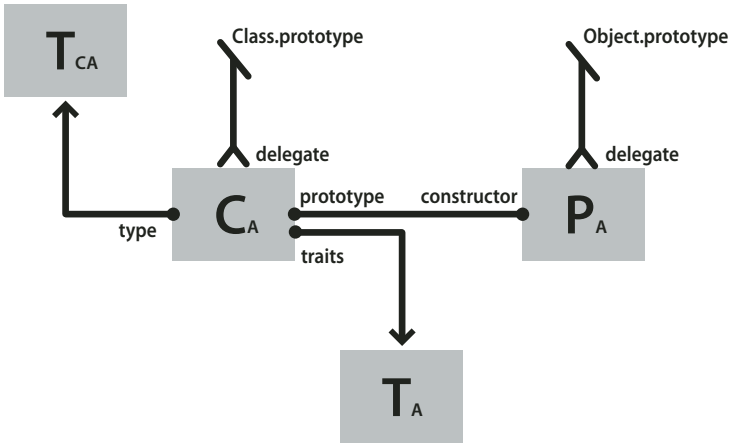
The constructor is now declared as part of the class definition, and the class properties `id` and `radius` must also be declared explicitly.

ActionScript 2.0 also added support for the definition of interfaces, which allow you to further refine your object-oriented programs with formally defined protocols for inter-object communication.

The ActionScript 3.0 class object

A common object-oriented programming paradigm, most commonly associated with Java and C++, uses classes to define types of objects. Programming languages that adopt this paradigm also tend to use classes to construct instances of the data type that the class defines. ActionScript uses classes for both of these purposes, but its roots as a prototype-based language add an interesting characteristic. ActionScript creates for each class definition a special class object that allows sharing of both behavior and state. For many ActionScript programmers, however, this distinction may have no practical coding implications. ActionScript 3.0 is designed such that you can create sophisticated object-oriented ActionScript applications without using, or even understanding, these special class objects.

The following diagram shows the structure of a class object that represents a simple class named A that is defined with the statement `class A {}`:



Each rectangle in the diagram represents an object. Each object in the diagram has a subscript character A to represent that it belongs to class A. The class object (CA) contains references to a number of other important objects. An instance traits object (TA) stores the instance properties that are defined within a class definition. A class traits object (TCA) represents the internal type of the class and stores the static properties defined by the class (the subscript character C stands for "class"). The prototype object (PA) always means the class object to which it was originally attached through the `constructor` property.

The traits object

The traits object, which is new in ActionScript 3.0, was implemented with performance in mind. In previous versions of ActionScript, name lookup could be a time-consuming process as Flash Player walked the prototype chain. In ActionScript 3.0, name lookup is much more efficient and less time consuming, because inherited properties are copied down from superclasses into the traits object of subclasses.

The traits object is not directly accessible to programmer code, but its presence can be felt by the improvements in performance and memory usage. The traits object provides the AVM2 with detailed information about the layout and contents of a class. With such knowledge, the AVM2 is able to significantly reduce execution time, because it can often generate direct machine instructions to access properties or call methods directly without a time-consuming name lookup.

Thanks to the traits object, an object's memory footprint can be significantly smaller than a similar object in previous versions of ActionScript. For example, if a class is sealed (that is, the class is not declared dynamic), an instance of the class does not need a hash table for dynamically added properties, and can hold little more than a pointer to the traits objects and some slots for the fixed properties defined in the class. As a result, an object that required 100 bytes of memory in ActionScript 2.0 could require as little as 20 bytes of memory in ActionScript 3.0.

Note: *The traits object is an internal implementation detail, and there is no guarantee that it will not change or even disappear in future versions of ActionScript.*

The prototype object

Every ActionScript class object has a property named `prototype`, which is a reference to the class's prototype object. The prototype object is a legacy of ActionScript's roots as prototype-based language. For more information, see History of ActionScript OOP support.

The `prototype` property is read-only, which means that it cannot be modified to point to different objects. This differs from the class `prototype` property in previous versions of ActionScript, where the prototype could be reassigned so that it pointed to a different class. Although the `prototype` property is read-only, the prototype object that it references is not. In other words, new properties can be added to the prototype object. Properties added to the prototype object are shared among all instances of the class.

The prototype chain, which was the only inheritance mechanism in previous versions of ActionScript, serves only a secondary role in ActionScript 3.0. The primary inheritance mechanism, fixed property inheritance, is handled internally by the traits object. A fixed property is a variable or method that is defined as part of a class definition. Fixed property inheritance is also called class inheritance, because it is the inheritance mechanism that is associated with keywords such as `class`, `extends`, and `override`.

The prototype chain provides an alternative inheritance mechanism that is more dynamic than fixed property inheritance. You can add properties to a class's prototype object not only as part of the class definition, but also at run time through the class object's `prototype` property. Note, however, that if you set the compiler to strict mode, you may not be able to access properties added to a prototype object unless you declare a class with the `dynamic` keyword.

A good example of a class with several properties attached to the prototype object is the Object class. The Object class's `toString()` and `valueOf()` methods are actually functions assigned to properties of the Object class's prototype object. The following is an example of how the declaration of these methods could, in theory, look (the actual implementation differs slightly because of implementation details):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

As mentioned previously, you can attach a property to a class's prototype object outside the class definition. For example, the `toString()` method can also be defined outside the Object class definition, as follows:

```
Object.prototype.toString = function()
{
    // statements
};
```

Unlike fixed property inheritance, however, prototype inheritance does not require the `override` keyword if you want to redefine a method in a subclass. For example. if you want to redefine the `valueOf()` method in a subclass of the Object class, you have three options. First, you can define a `valueOf()` method on the subclass's prototype object inside the class definition. The following code creates a subclass of Object named Foo and redefines the `valueOf()` method on Foo's prototype object as part of the class definition. Because every class inherits from Object, it is not necessary to use the `extends` keyword.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

Second, you can define a `valueOf()` method on Foo's prototype object outside the class definition, as shown in the following code:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Third, you can define a fixed property named `valueOf()` as part of the Foo class. This technique differs from the others in that it mixes fixed property inheritance with prototype inheritance. Any subclass of Foo that wants to redefine `valueOf()` must use the `override` keyword. The following code shows `valueOf()` defined as a fixed property in Foo:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

The AS3 namespace

The existence of two separate inheritance mechanisms, fixed property inheritance and prototype inheritance, creates an interesting compatibility challenge with respect to the properties and methods of the core classes. Compatibility with the ECMAScript language specification on which ActionScript is based requires the use of prototype inheritance, which means that the properties and methods of a core class are defined on the prototype object of that class. On the other hand, compatibility with ActionScript 3.0 calls for the use of fixed property inheritance, which means that the properties and methods of a core class are defined in the class definition using the `const`, `var`, and `function` keywords. Moreover, the use of fixed properties instead of the prototype versions can lead to significant increases in run-time performance.

ActionScript 3.0 solves this problem by using both prototype inheritance and fixed property inheritance for the core classes. Each core class contains two sets of properties and methods. One set is defined on the prototype object for compatibility with the ECMAScript specification, and the other set is defined with fixed properties and the AS3 namespace for compatibility with ActionScript 3.0.

The AS3 namespace provides a convenient mechanism for choosing between the two sets of properties and methods. If you do not use the AS3 namespace, an instance of a core class inherits the properties and methods defined on the core class's prototype object. If you decide to use the AS3 namespace, an instance of a core class inherits the AS3 versions because fixed properties are always preferred over prototype properties. In other words, whenever a fixed property is available, it is always used instead of an identically named prototype property.

You can selectively use the AS3 namespace version of a property or method by qualifying it with the AS3 namespace. For example, the following code uses the AS3 version of the `Array.pop()` method:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Alternatively, you can use the `use namespace` directive to open the AS3 namespace for all the definitions within a block of code. For example, the following code uses the `use namespace` directive to open the AS3 namespace for both the `pop()` and `push()` methods:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 also provides compiler options for each set of properties so that you can apply the AS3 namespace to your entire program. The `-as3` compiler option represents the AS3 namespace, and the `-es` compiler option represents the prototype inheritance option (`es` stands for ECMAScript). To open the AS3 namespace for your entire program, set the `-as3` compiler option to `true` and the `-es` compiler option to `false`. To use the prototype versions, set the compiler options to the opposite values. The default compiler settings for Flash Builder and Flash Professional are `-as3 = true` and `-es = false`.

If you plan to extend any of the core classes and override any methods, you should understand how the AS3 namespace can affect how you must declare an overridden method. If you are using the AS3 namespace, any method override of a core class method must also use the AS3 namespace along with the `override` attribute. If you are not using the AS3 namespace and want to redefine a core class method in a subclass, you should not use the AS3 namespace or the `override` keyword.

Example: GeometricShapes

The GeometricShapes sample application shows how a number of object-oriented concepts and features can be applied using ActionScript 3.0, including:

- Defining classes
- Extending classes
- Polymorphism and the `override` keyword
- Defining, extending, and implementing interfaces

It also includes a "factory method" that creates class instances, showing how to declare a return value as an instance of an interface, and use that returned object in a generic way.

To get the application files for this sample, see www.adobe.com/go/learn_programmingAS3samples_flash. The GeometricShapes application files can be found in the folder Samples/GeometricShapes. The application consists of the following files:

File	Description
GeometricShapes.mxml or GeometricShapes.fla	The main application file in Flash (FLA) or Flex (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	The base interface defining methods to be implemented by all GeometricShapes application classes.
com/example/programmingas3/geometricshapes/IPolygon.as	An interface defining methods to be implemented by GeometricShapes application classes that have multiple sides.
com/example/programmingas3/geometricshapes/RegularPolygon.as	A type of geometric shape that has sides of equal length postponed symmetrically around the shape's center.
com/example/programmingas3/geometricshapes/Circle.as	A type of geometric shape that defines a circle.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	A subclass of RegularPolygon that defines a triangle with all sides the same length.
com/example/programmingas3/geometricshapes/Square.as	A subclass of RegularPolygon defining a rectangle with all four sides the same length.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	A class containing a factory method for creating shapes given a shape type and size.

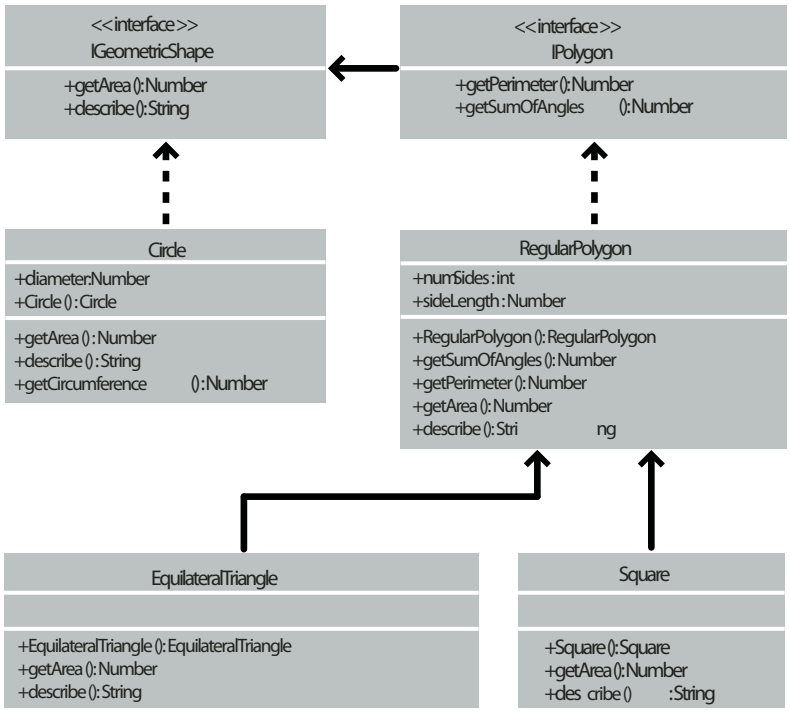
Defining the GeometricShapes classes

The GeometricShapes application lets the user specify a type of geometric shape and a size. It then responds with a description of the shape, its area, and distance around its perimeter.

The application user interface is trivial, including a few controls for selecting the type of shape, setting the size, and displaying the description. The most interesting part of this application is under the surface, in the structure of the classes and interfaces themselves.

This application deals with geometric shapes, but it doesn't display them graphically.

The classes and interfaces that define the geometric shapes in this example are shown in the following diagram using Unified Modeling Language (UML) notation:



GeometricShapes Example Classes

Defining common behavior with interfaces

This GeometricShapes application deals with three types of shapes: circles, squares, and equilateral triangles. The GeometricShapes class structure begins with a very simple interface, IGeometricShape, that lists methods common to all three types of shapes:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

The interface defines two methods: the `getArea()` method, which calculates and returns the area of the shape, and the `describe()` method, which assembles a text description of the shape's properties.

It's also desirable to know the distance around the perimeter of each shape. However, the perimeter of a circle is called the circumference, and it's calculated in a unique way, so the behavior diverges from that of a triangle or a square. Still there is enough similarity between triangles, squares, and other polygons that it makes sense to define a new interface class just for them: IPolygon. The IPolygon interface is also rather simple, as shown here:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

This interface defines two methods common to all polygons: the `getPerimeter()` method that measures the combined distance of all the sides, and the `getSumOfAngles()` method that adds up all the interior angles.

The IPolygon interface extends the IGeometricShape interface, which means that any class that implements the IPolygon interface must declare all four methods—the two from the IGeometricShape interface, and the two from the IPolygon interface.

Defining the shape classes

Once you have a good idea about the methods common to each type of shape, you can define the shape classes themselves. In terms of how many methods you need to implement, the simplest shape is the Circle class, shown here:

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius * radius.
            var radius:Number = diameter / 2;
            return Math.PI * radius * radius;
        }

        public function getCircumference():Number
        {
            // The formula is Pi * diameter.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

The Circle class implements the IGeometricShape interface, so it must provide code for both the `getArea()` method and the `describe()` method. In addition, it defines the `getCircumference()` method, which is unique to the Circle class. The Circle class also declares a property, `diameter`, which won't be found in the other polygon classes.

The other two types of shapes, squares and equilateral triangles, have some other things in common: they each have sides of equal length, and there are common formulas you can use to calculate the perimeter and sum of interior angles for both. In fact, those common formulas apply to any other regular polygons that you define in the future as well.

The RegularPolygon class is the superclass for both the Square class and the EquilateralTriangle class. A superclass lets you define common methods in one place, so you don't have to define them separately in each subclass. Here is the code for the RegularPolygon class:

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + "
degrees.\n";
            return desc;
        }
    }
}
```

First, the RegularPolygon class declares two properties that are common to all regular polygons: the length of each side (the `sideLength` property) and the number of sides (the `numSides` property).

The RegularPolygon class implements the IPolygon interface and declares all four of the IPolygon interface methods. It implements two of these—the `getPerimeter()` and `getSumOfAngles()` methods—using common formulas.

Because the formula for the `getArea()` method differs from shape to shape, the base class version of the method cannot include common logic that can be inherited by the subclass methods. Instead, it simply returns a 0 default value to indicate that the area was not calculated. To calculate the area of each shape correctly, the subclasses of the RegularPolygon class have to override the `getArea()` method themselves.

The following code for the EquilateralTriangle class show how the `getArea()` method is overridden:

```
package com.example.programmingas3.geometricshapes
{
    public class EquilateralTriangle extends RegularPolygon
    {
        public function EquilateralTriangle(len:Number = 100):void
        {
            super(len, 3);
        }

        public override function getArea():Number
        {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
        }

        public override function describe():String
        {
                /* starts with the name of the shape, then delegates the rest
                of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
        }
    }
}
```

The `override` keyword indicates that the `EquilateralTriangle.getArea()` method intentionally overrides the `getArea()` method from the RegularPolygon superclass. When the `EquilateralTriangle.getArea()` method is called, it calculates the area using the formula in the preceding code, and the code in the `RegularPolygon.getArea()` method never executes.

In contrast, the EquilateralTriangle class doesn't define its own version of the `getPerimeter()` method. When the `EquilateralTriangle.getPerimeter()` method is called, the call goes up the inheritance chain and executes the code in the `getPerimeter()` method of the RegularPolygon superclass.

The `EquilateralTriangle()` constructor uses the `super()` statement to explicitly invoke the `RegularPolygon()` constructor of its superclass. If both constructors had the same set of parameters, you could have omitted the `EquilateralTriangle()` constructor completely, and the `RegularPolygon()` constructor would be executed instead. However, the `RegularPolygon()` constructor needs an extra parameter, `numSides`. So the `EquilateralTriangle()` constructor calls `super(len, 3)`, which passes along the `len` input parameter and the value 3 to indicate that the triangle has three sides.

The `describe()` method also uses the `super()` statement, but in a different way. It uses it to invoke the RegularPolygon superclass' version of the `describe()` method. The `EquilateralTriangle.describe()` method first sets the `desc` string variable to a statement about the type of shape. Then it gets the results of the `RegularPolygon.describe()` method by calling `super.describe()`, and it appends that result to the `desc` string.

The Square class isn't described in detail here, but it is similar to the EquilateralTriangle class, providing a constructor and its own implementations of the `getArea()` and `describe()` methods.

Polymorphism and the factory method

A set of classes that make good use of interfaces and inheritance can be used in many interesting ways. For example, all of the shape classes described so far either implement the IGeometricShape interface or extend a superclass that does. So if you define a variable to be an instance of IGeometricShape, you don't have to know whether it is actually an instance of the Circle or the Square class to call its `describe()` method.

The following code shows how this works:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

When `myShape.describe()` is called, it executes the method `Circle.describe()`, because even though the variable is defined as an instance of the IGeometricShape interface, Circle is its underlying class.

This example shows the principle of polymorphism in action: the exact same method call results in different code being executed, depending on the class of the object whose method is being invoked.

The GeometricShapes application applies this kind of interface-based polymorphism using a simplified version of a design pattern known as the factory method. The term *factory method* means a function that returns an object whose underlying data type or contents can differ depending on the context.

The GeometricShapeFactory class shown here defines a factory method named `createShape()`:

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

The `createShape()` factory method lets the shape subclass constructors define the details of the instances that they create, while returning the new objects as IGeometricShape instances so that they can be handled by the application in a more general way.

The `describeShape()` method in the preceding example shows how an application can use the factory method to get a generic reference to a more specific object. The application can get the description for a newly created Circle object like this:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

The `describeShape()` method then calls the `createShape()` factory method with the same parameters, storing the new Circle object in a static variable named `currentShape`, which was typed as an IGeometricShape object. Next, the `describe()` method is called on the `currentShape` object, and that call is automatically resolved to execute the `Circle.describe()` method, returning a detailed description of the circle.

Enhancing the sample application

The real power of interfaces and inheritance becomes apparent when you enhance or change your application.

Say that you wanted to add a new shape, a pentagon, to this sample application. You would create a Pentagon class that extends the RegularPolygon class and defines its own versions of the `getArea()` and `describe()` methods. Then you would add a new Pentagon option to the combo box in the application's user interface. But that's it. The Pentagon class would automatically get the functionality of the `getPerimeter()` method and the `getSumOfAngles()` method from the RegularPolygon class by inheritance. Because it inherits from a class that implements the IGeometricShape interface, a Pentagon instance can be treated as an IGeometricShape instance too. That means that to add a new type of shape, you do not need to change the method signature of any of the methods in the GeometricShapeFactory class (and consequently, you don't need to change any of the code that uses the GeometricShapeFactory class either).

You may want to add a Pentagon class to the Geometric Shapes example as an exercise, to see how interfaces and inheritance can ease the workload of adding new features to an application.